UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Douglas Nascimento Rechia

ESPECIFICAÇÃO FORMAL DE RESTRIÇÕES DE PROJETO PARA *FRAMEWORKS* ORIENTADOS A OBJETOS

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação

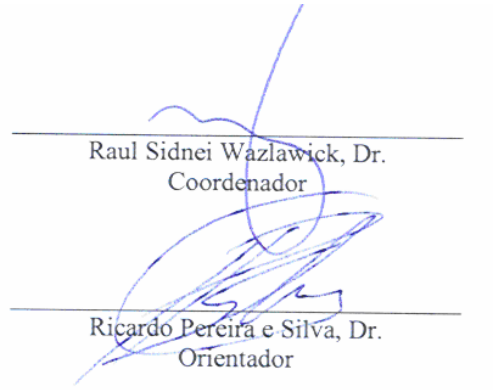
Ricardo Pereira e Silva
Orientador

Florianópolis, dezembro de 2005

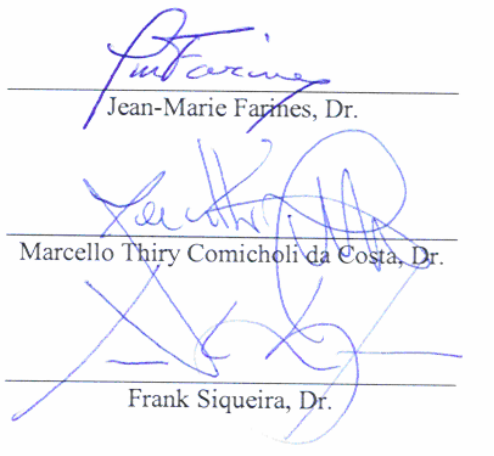
ESPECIFICAÇÃO FORMAL DE RESTRIÇÕES DE PROJETO PARA FRAMEWORKS ORIENTADOS A OBJETOS

Douglas Nascimento Rechia

Esta dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação Área de Concentração Sistemas de Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.



Banca examinadora



“From now on we live in a world where man has walked on the Moon. It's not a miracle; we just decided to go.” – Tom Hanks

AGRADECIMENTOS

Agradeço ao Professor Ricardo Pereira e Silva pela orientação, incentivo, e valiosas contribuições. Agradeço também aos professores Jean-Marie Farines, Marcello Thiry e Frank Siqueira, que aceitaram o convite para formar a banca avaliadora desta pesquisa.

Ao Professor Ricardo Felipe Custódio por me ajudar a não deixar os objetivos acadêmicos se perderem de vista.

À Motorola Industrial Ltda e a todos os colegas de trabalho do projeto *Test Automation*, especialmente a Valdir Reis, Luiz Kawakami e André Knabben por terem confiado a mim esta pesquisa, e pelas diferentes contribuições.

A todos os colegas do Labsoft pela paciência, confiança e determinação em alcançar o êxito, em especial a Otávio Pereira pelo auxílio para lidar com Aspectos, e a Roberto Cunha, pelas dicas e referências com redes de Petri.

A todos os amigos, em especial a Vera, Guilherme, Laurinha, Maureci, Thiago, Graziele, Daniele, Tábita, Faylane.

À minha namorada Caroline, por me fazer muito mais feliz, por me fazer uma pessoa melhor, por me ajudar a estar ainda mais perto de Jeová. Mesmo não sabendo disso, ajudaste-me a concluir este trabalho com sucesso.

À minha família, meu irmão Diego, e especialmente aos meus pais Valtenor e Leni, pelo incentivo e ajuda incondicional, pela compreensão na ausência de casa nas horas de dedicação aos estudos, e pelo apoio emocional para conquistar mais este desafio. Sem vocês, nunca teria conseguido.

A Jeová Deus, aquele que é “digno ... de receber a glória, e a honra, e o poder, porque criaste todas as coisas e porque elas existiram e foram criadas por tua vontade”. – Revelação 4:11

SUMÁRIO

LISTA DE ACRÔNIMOS

AFD – Autômato Finito Determinístico

AFN – Autômato Finito Não-determinístico

AOP – *Aspect-Oriented Programming*

ATM – *Automatic Teller Machine*

BOON – *Basic Object-Oriented Notation*

FCL – *Framework Constraints Language*

MFC – *Microsoft Foundation Classes*

MIT – *Massachussetts Institute of Technology*

PTF – *Phone Test Framework*

TAF – *Test Automation Framework*

UF – *Utility Function*

UML – *Unified Modeling Language*

USB – *Universal Serial Bus*

XML – *eXtensible Markup Language*

LISTA DE FIGURAS

RESUMO

Um dos desafios encontrados para a concepção de sistemas a partir de *frameworks* orientados a objetos é a sua adequada utilização. Utilizar um *framework* adequadamente para constituir um sistema implica em obedecer às suas restrições, definidas em seu projeto. Dentre as restrições que o desenvolvedor da aplicação sob um *framework* precisa observar, está a correta ordem de execução de métodos dos objetos disponibilizados pelo *framework*. Nesse contexto, esse trabalho propõe uma maneira de especificar formalmente as restrições que um *framework* impõe em termos da ordem em que seus métodos podem ser invocados. A partir da especificação formal, gera-se automaticamente código, baseado no paradigma de desenvolvimento orientado a aspectos, capaz de verificar se as restrições formalmente especificadas estão sendo observadas em tempo de execução.

De forma transparente ao usuário do *framework* e ao desenvolvedor da aplicação final, o código automaticamente gerado instancia uma rede de Petri, a qual será responsável pelo controle do estado de cada objeto especificado e o controle do estado do sistema. Ambos os desenvolvedores são beneficiados: o desenvolvedor do *framework* não precisa escrever código que testa as restrições do *framework* e o desenvolvedor da aplicação final será notificado se ele estiver fazendo invocações indevidas a um método.

A abordagem formal proposta neste trabalho é aplicada ao *TAF*, um *framework* desenvolvido pela Motorola para a criação de casos de teste automatizados para telefones celulares.

Palavras-chave: Engenharia de *software*, Verificação de *software*/programas, Verificadores de asserções, Métodos formais, Verificação de modelo, Confiabilidade, Validação.

ABSTRACT

One of the challenges found when creating computer programs under object-oriented frameworks is its correct use. To use a framework correctly in order to create a complete system, it is needed to meet its design restrictions. One of such restrictions is the correct order the methods provided by the framework are invoked. In this context, this work presents a way to formally specify the methods call order restrictions that a framework may have. As a consequence of the formal specification, code is automatically generated, based on aspect-oriented paradigm, which is able to check if the restrictions formally stated are being accomplished at run time.

The automatically generated code instantiates internally a Petri Net which is responsible for keeping track of both system's state and the state of each object that was previously specified. The formal model internally used, the Petri nets, is transparent for the framework developer and the developer of the complete system. Both are benefited: the framework developer does not need to write code to verify the framework restrictions, and the developer of the complete system will be notified in case of any method is being called in a wrong place.

The approach proposed in this work is applied in TAF, a framework developed by Motorola, which was created to automate test cases for mobile phones.

***Keywords**: Software engineering, Software/Program verification, Assertion checkers, Formal methods, Model checking, Reliability, Validation*

1 INTRODUÇÃO

A concepção de sistemas a partir da extensão de *frameworks* orientados a objetos tem sido crescente a partir da década de 1990. São várias as razões pelas quais os *frameworks*[1] tornaram-se mais populares e chamaram a atenção da comunidade científica a partir de então. A reutilização de código e projeto proporcionado pelos *frameworks* podem reduzir consideravelmente o tempo gasto para a construção de um sistema completo que atenda a requisitos funcionais (MILI et al., 1995; MOSER & NIERSTRASZ, 1993). Robustez, flexibilidade à mudança de requisitos e arquitetura são outras características que fomentam a utilização de *frameworks* para a criação de *software*.

Um obstáculo ao uso dos *frameworks* é a sua correta utilização. Para estender corretamente um *framework* a fim de criar um sistema completo, é preciso obedecer às suas *restrições*. Segundo STACY et al. (1992) apud HOU & HOOVER (2001), o usuário de um *framework* precisará respeitar, dentre outras, as seguintes restrições a fim de usá-lo:

a) garantir que quaisquer métodos reimplementados não violem os contratos que o *framework* assume;
b) respeitar os protocolos suportados pelas superclasses;
c) criar objetos com os parâmetros corretos;
d) prover respostas apropriadas às chamadas feitas pelos objetos do *framework*; e
e) respeitar a sequência de operações que podem ser executadas em um objeto específico.

As restrições ao uso de *frameworks* são estabelecidas geralmente de maneira informal, e não raro disponibilizadas com sua documentação, assim como sugerido por MURRAY et al. (2004). Entretanto, descrições textuais informais podem ser ambíguas e podem facilmente conduzir a erros. Além disso, descrições textuais não podem ser

[1] No contexto desse trabalho, o termo *framework* é utilizado sempre para referir-se aos *frameworks* orientados a objeto.

automaticamente verificadas.

O emprego de métodos formais pode auxiliar o usuário de um *framework* a desenvolver aplicações sobre este *framework* de maneira adequada. A especificação formal proporciona a verificação automática da aplicação criada sobre o *framework*, indicando ao desenvolvedor qualquer inconsistência encontrada. A partir de uma descrição formal de classes, o desenvolvedor poderá assegurar-se de que seu sistema se comportará conforme o esperado, e qualquer efeito colateral decorrente da má utilização do *framework* será evitado.

Métodos formais como *Lambda Calculus* têm sido usados para especificar *frameworks* (CAMPO, 97). Estes, entretanto, produzem na maioria das vezes descrições longas e de baixo nível, difíceis de compreender e de serem aplicadas a um sistema extenso. Para contornar esse problema, SILVA (2000) propôs um ambiente de suporte ao desenvolvimento e uso de *frameworks* chamado SEA, no qual técnicas de modelagem gráficas foram usadas para a descrição de *frameworks*. Silva argumenta que o uso de notações gráficas para a criação de um modelo auxilia o "processo mental de criação de *frameworks"*. O ambiente SEA suporta também a validação da compatibilidade estrutural e comportamental entre componentes para a composição de um sistema, mas não provê nenhum mecanismo para a validação de *frameworks*. No SEA, a análise de compatibilidade de componentes é solucionada através de um modelo formal, as redes de Petri, que possuem tanto uma representação gráfica quanto uma representação algébrica, alcançando assim a legibilidade proporcionada pelo grafo das redes de Petri e o formalismo que fundamenta a teoria deste modelo. Uma combinação da abordagem de SILVA (2000) para a validação de componentes com a especificação formal de *frameworks* permitiria especificar um *framework* por completo, sob o ponto de vista da adequação do seu uso às suas restrições de projeto.

Dentre as restrições listadas acima, as quais o usuário de um *framework* precisará respeitar a fim de produzir a aplicação final, este trabalho propõe um método para formalizar as restrições associadas às classes que compõem um *framework* evitando que os métodos dos objetos disponibilizados por este sejam invocados na ordem incorreta. Somente será requerido do projetista do *framework* que as restrições ao seu uso sejam especificadas de acordo com uma sintaxe específica. Esta especificação possibilitará a geração automática de código que garantirá que qualquer aplicação

contruída sobre o *framework* respeite a ordem de chamada de seus métodos. O projetista do *framework* assim ficará livre da tarefa de fazer este tipo de verificação, e o usuário do *framework* terá a garantia de que o seu sistema não viola nenhuma restrição imposta pelo *framework* no sentido da ordem de chamada de métodos. O método formal descrito neste trabalho é projetado de forma que seja fácil de utilizar, legível e utilizado como complemento a linguagens de modelagem gráficas populares, como UML (OMG, 2005) e as extensões de UML propostas por SILVA (2000).

1.1 Objetivos

1.1.1 Objetivo geral

Prover uma maneira de especificar formalmente as restrições de ordem de chamada de métodos de classes que compõem um *framework,* de forma que seja possível verificar se o mesmo está sendo utilizado de maneira correta, sem aumentar a complexidade de uso sob o ponto de vista do usuário.

1.1.2 Objetivos específicos

a) Definir como especificar formalmente as classes de um *framework* de forma que o seu projetista possa fazê-lo mesmo sem ser um especialista em métodos formais.
b) Gerar código automaticamente a partir da especificação formal de um *framework*. Este código deverá verificar se os sistemas construídos sob o *framework* o estão utilizando de maneira apropriada.
c) Diminuir o tempo gasto para a criação de um sistema a partir de um *framework*, evitando atrasos decorrentes de seu mau uso.
d) Diminuir falhas e defeitos de sistemas construídos a partir de *frameworks*.
e) Manter o controle dos métodos invocados dos objetos de um *framework*. Isto possibilitará ao *framework* impedir que um método específico seja chamado no momento errado.

1.2 Metodologia

A fim de alcançar os objetivos estabelecidos neste trabalho, aplicam-se os seguintes métodos:

a) Utilização de um modelo formal para especificação de *frameworks* orientados a objetos – definição de forma de uso das redes de Petri de maneira que as restrições de ordem de chamada de métodos de um objeto sejam escritas de acordo com este padrão.
b) Geração de código automática – a partir da especificação formal de classes segundo o método proposto, gera-se código que impede a utilização incorreta de objetos desta classe.
c) Aplicação do método concebido em um *framework* utilizado no meio corporativo – a utilização do mecanismo proposto em uma aplicação do mundo real é a forma de validar o modelo.

1.3 Justificativa

A falta de formalismo e a utilização de linguagem natural para a documentação de artefatos de software não é suficiente para garantir a correta utilização destes. Embora os *frameworks* constituam uma das formas mais promissoras de promover o reuso de código para a concepção de sistemas, a reutilização esperada poderá não ser alcançada se métodos formais não forem aplicados (HOU & HOOVER, 2001). Segundo FONTOURA et al. (2000) a instanciação de um framework para a composição de uma aplicação utilizando apenas a linguagem-alvo na qual o *framework* foi escrito não pode verificar qualquer restrição de instanciação, logo nenhuma mensagem de erro é mostrada ao usuário do *framework*. Ainda segundo OLIVEIRA et al. (2004), documentação informal não é suficiente para verificar se o projeto final do sistema está de acordo com as premissas originais do *framework*.

Dadas as falhas observadas pela especificação de *frameworks* via métodos não formais, este trabalho justifica-se pela utilização de métodos formais para garantir a sua correta utilização. Os métodos formais possibilitarão também a geração automática de código e a verificação automática do sistema, isto é, sua conformidade com a especificação.

1.4 Resultados esperados

Espera-se que a utilização de métodos formais em *softwares* criados com o auxílio de *frameworks* gere sistemas mais confiáveis, menos sujeitos a defeitos e de melhor qualidade. O tempo de desenvolvimento de sistemas suportados por *frameworks* deve diminuir, uma vez que o formalismo apontará ao programador aonde está a inconsistência ainda em tempo de desenvolvimento (isto é, a não-conformidade com a especificação do *framework* é apontada). Sistemas críticos tais como os de monitoramento de aparelhos médicos, sistemas de aviação, etc que simplesmente não podem falhar poderão tornar-se mais seguros, apresentando o comportamento real esperado.

1.5 Estrutura da dissertação

Esta dissertação é composta pelo presente capítulo introdutório, quatro capítulos de desenvolvimento, um capítulo de conclusão, e quatro anexos.

O capítulo 2 apresenta os principais conceitos consolidados e preliminares relativos à criação e ao uso de *frameworks* orientados a objetos, redes de Petri e Programação Orientada a Aspectos.

O capítulo 3 mostra o estado da arte da especificação formal aplicada à validação de sistemas construídos sob *frameworks* orientados a objetos.

O capítulo 4 apresenta as principais contribuições desse trabalho de pesquisa: a criação de um esquema XML para especificação de *frameworks* orientados a objetos, a utilização de máquinas de estado para controlar o estado de objetos, a criação de uma rede de Petri para controlar o estado do sistema a partir das máquinas de estado criadas para cada objeto, e a geração de código automática baseado em aspectos que valida a aplicação construída sob o *framework*. Este capítulo exemplifica a aplicação dos métodos utilizados tomando como base um *framework* fictício utilizado por outro trabalho de pesquisa em métodos formais.

No capítulo 5 são aplicados os conceitos e o método formal proposto no capítulo 4 no TAF, um *framework* usado no meio corporativo para a automação de casos de teste.

No capítulo 6 são mostradas as conclusões constatadas por esse trabalho de pesquisa.

O Anexo 1 lista o esquema XML utilizado para reger a especificação formal proposta por este trabalho. O Anexo 2 lista o código automaticamente gerado pela especificação do *framework*-exemplo utilizado no Capítulo 4. O Anexo 3 lista um caso de teste construído sob o TAF; este caso de teste é validado utilizando o código automaticamente gerado pela ferramenta produzida neste trabalho. O Anexo 4 lista a especificação formal XML do TAF que valida o caso de teste listado no Anexo 3.

2 CONCEITOS TRATADOS NO CONTEXTO DO TRABALHO DE PESQUISA

Apresentam-se nesse capítulo alguns conceitos preliminares que são aplicados no desenvolvimento da presente pesquisa. Os conceitos abordados são relativos a: *frameworks* orientados a objetos, redes de Petri e Programação Orientada a Aspectos.

2.1 *Frameworks* orientados a objetos

Um *framework* orientado a objetos é uma estrutura de classes projetada para generalizar um domínio de aplicações. Segundo JOHNSON & FOOTE (1998), um *framework* é "um conjunto de classes que contém o projeto abstrato de soluções para uma família de problemas relacionados, e que suporta alto nível de reutilização com alta granularidade". Em outras palavras, trata-se de um projeto incompleto produzido para, futuramente, gerar aplicações funcionais que estejam no domínio do problema ao qual o *framework* está inserido. A principal motivação para o uso de *frameworks* para a concepção de sistemas é a reutilização de código e a diminuição do tempo gasto para implementar a aplicação final.

A estrutura de classes de um *framework* orientado a objetos é tipicamente escrita com classes abstratas. Visto que uma classe abstrata não possui instâncias, ela é freqüentemente utilizada como um modelo para a criação de classes concretas. Os *frameworks* geralmente utilizam classes abstratas, pois essas definem a interface de componentes e provêem um esqueleto que pode ser extendido pela aplicação final (FAYAD et al., 1999). As classes abstratas de um *framework* geralmente possuem métodos concretos e métodos abstratos. O desenvolvedor da aplicação construída sob o *framework* especificará o comportamento específico de sua aplicação criando classes concretas que podem reimplementar métodos concretos e implementam os métodos abstratos. Os métodos que o desenvolvedor da aplicação poderá ou deverá implementar (as partes flexíveis) são chamados de *hot spots*.

Uma característica dos *frameworks* orientados a objetos é o fato de que o próprio *framework* é quem determina o fluxo de execução do programa. Em outras palavras, o *framework* é quem chama os métodos criados pelo desenvolvedor da

aplicação final, e não ao contrário, como tradicionalmente ocore quando se utilizam bibliotecas de classes. O fato de que os *frameworks* chamam métodos, ao contrário de serem chamados, é chamado de *Princípio de Hollywood* (BOSCH et al., 1999). A Figura 2.1 mostra esquematicamente a forma mais comum de arquitetura de uma aplicação contruída com um *framework*. As classes destacadas no retângulo cinza são as classes do *framework*, e as classes fora deste retângulo são as classes concretas implementadas para produzir a aplicação final.

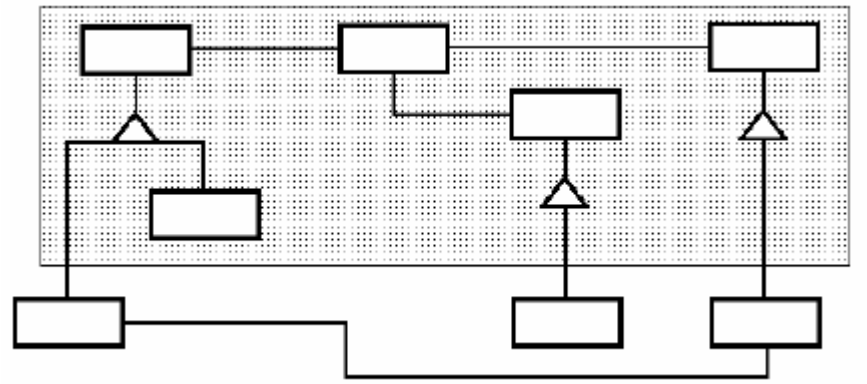


Figura 2.1: Aplicação desenvolvida reutilizando um *framework*. Fonte: SILVA (2000)

2.1.1 Uso de *frameworks* orientados a objeto

Segundo FAYAD et al. (1999), existem basicamente três maneiras de usar um *framework*. A primeira maneira, e a mais fácil, consiste em conectar objetos que o *framework* provê, por composição. Não são criadas subclasses das classes do *framework*, nem tampouco o *framework* propriamente dito é alterado. Apenas as interfaces do *framework* e suas regras de composição de objetos são utilizadas, assim como se conectam circuitos em uma placa. O programador que utiliza o *framework* dessa forma não necessita conhecer os tipos dos objetos nem a especificação exata destes para fazer as ligações.

A segunda maneira de utilizar um *framework* é pela criação de subclasses concretas a partir da especialização de classes do próprio *framework* (FAYAD et al. 1999). As subclasses concretas são utilizadas para a definição do comportamento específico da aplicação que está sendo construída. Como a criação de subclasses via herança não é trivial e gera classes fortemente acopladas às classes do *framework*, é

necessário que o desenvolvedor conheça melhor o próprio *framework*, as interfaces de seus componentes e suas restrições.

Ainda de acordo com FAYAD et al. (1999), a terceira forma de usar um *framework* consiste em alterar as classes abstratas que formam o "coração" do *framework*. Esta é a maneira mais difícil, e que requer ainda mais conhecimento do usuário referente ao projeto e arquitetura utilizados na criação do *framework*. Geralmente, acrescentam-se novos métodos ou atributos às classes já existentes. Será necessário, portanto, obter e alterar o código fonte original. A recompensa é a possibilidade de uso mais flexível e poderoso. Por outro lado, alterações nas classes de um *framework* poderão quebrar o código das subclasses ou mesmo o código das aplicações já existentes.

Se o programador da aplicação puder usar um *framework* sem conhecer seus componentes e arquitetura, isto é, se ele puder usá-lo somente através da composição de objetos, então o *framework* é classificado como *caixa preta* (FAYAD et al. 1999). Se a utilização for baseada apenas no reuso através de herança – para originar subclasses concretas para a criação da aplicação final – o *framework* é classificado como *caixa branca*. Uma combinação das duas abordagens pode ser utilizada, o que corresponderia a um *framework caixa cinza* (SILVA, 2000).

2.1.1.1 Restrições ao uso de *frameworks*

Conforme já parcialmente abordado no capítulo introdutório, é preciso obedecer às restrições de um *framework* para que este possa ser corretamente usado para criar a aplicação final – restrições essas estabelecidas pelo desenvolvedor do *framework*, em termos do que é e do que não é flexível. O uso de um *framework* envolve, no caso mais geral dos caixa-cinza, a conexão de objetos (via composição) e a criação de classes concretas, através da especialização das classes do *framework*.

Segundo STACY et al. (1992), poderá ser exigido do desenvolvedor que está reutilizando objetos por composição o seguinte:

> *"a) criar objetos com os parâmetros corretos e compô-los corretamente somente com objetos compatíveis;*
> *b) respeitar a sequência de operações que podem ser executadas em um objeto específico;*
> *c) prover respostas apropriadas às chamadas feitas pelos objetos do framework."*

Quanto à reutilização de classes via especialização (herança), o desenvolvedor deverá observar o seguinte (STACY et al., 1992):

> *"a) implementar os métodos abstratos projetados para serem implementados pelas subclasses concretas;*
> *b) garantir que quaisquer métodos sobrescritos (overriden) não violem as asserções assumidas com relação a estes métodos;*
> *c) respeitar os protocolos suportados pelas superclasses;*
> *d) iniciar ações de notificação quando o estado da subclasse é alterado."*

STACY et al. (1992) menciona que a observância das restrições listadas acima garante a integridade semântica das classes que estão sendo reutilizadas, as quais, no caso do desenvolvimento com o uso de *frameworks*, são as classes que este provê. Algumas medidas são apontadas por Stacy para resolver o problema da integridade semântica:

a) realizar o projeto das classes que serão reutilizadas para conduzir ao menor número de erros possíveis por parte de seus usuários;
b) utilizar composição tanto quanto possível (herança geralmente utiliza abstrações de baixo nível, o que pode conduzir a erros com mais facilidade);
c) utilização de contratos;
d) uso de linguagens de descrição de interfaces e protocolos.

Observa-se, portanto, que é necessário que o usuário de um *framework* aprenda a utilizá-lo bem, isto é, esteja ciente de suas restrições e as respeite. Quanto mais o desenvolvedor desejar especializar o *framework*, mais ele estará suscetível a quebrar as restrições que o projeto original do *framework* impõe. A observância dessas restrições é indispensável para que as aplicações construídas sob o *framework* em questão apresentem o comportamento desejado. Uma vez vencida essa barreira, o uso de um *framework* poderá promover significativa reutilização de código, em vista do desenvolvimento de aplicações em menor tempo.

2.2 Redes de Petri

Carl Petri, em sua tese de doutorado entitulada *Comunicação com autômatos*

defendida em 1962 na Universidade de Darmstadt, Alemanha, propôs um sistema formal para modelar a comunicação entre autômatos finitos em sistemas discretos. Segundo CARDOSO & VALETTE (1997), a tese de Petri impulsionou pesquisadores do MIT (*Massachussetts Institute of Technology*) a conceberem o que mais tarde se tornou as *Redes de Petri*.

Neste trabalho, utilizam-se as redes de Petri como um mecanismo formal de validação do uso de um *framework* orientado a objetos. A seguir, são apresentados informalmente os principais elementos que compõem este modelo. Na seqüência, formaliza-se algebricamente o modelo matemático.

2.2.1 Apresentação informal das redes de Petri

A rede de Petri é um mecanismo criado para modelar Sistemas a Eventos Discretos[1]. Este mecanismo tem sido utilizado para modelar conceitos de processos em geral, como paralelismo, cooperação e competição. As redes de Petri foram criadas para superar limitações conhecidas das Máquinas de Estado, as quais são incapazes de modelar algumas seqüências de eventos.

São quatro os elementos básicos que definem uma rede de Petri:

a) *lugar* (graficamente representado com um círculo) – pode representar, dentre outras coisas, uma condição ou estado parcial. Em geral, cada lugar tem um predicado associado, como, por exemplo, *máquina livre*, *peça em espera*, etc.

b) *transição* (graficamente representado com uma barra ou retângulo) – representa um evento que ocorre no sistema;

c) *ficha* (representado graficamente com ponto dentro de um lugar) – trata-se de um indicador, significando que a condição associada ao lugar é verificada.

d) *arco* (representado graficamente com uma seta) – define a relação entre os lugares e as transições; os arcos colocadas no sentido lugar-transição definem os lugares de entrada desta transição, enquanto que

[1] Sistemas a Eventos Discretos são sistemas dirigidos por eventos, isto é, as mudanças de estado ocorrem por ocasião da ocorrência de eventos.

os arcos no sentido transição-lugar indicam os lugares de saída.

O estado do sistema é dado pela repartição de fichas nos lugares da rede de Petri. Associa-se um evento do sistema a cada transição. A ocorrência de um evento desencadeia o disparo da transição associada e, conseqüentemente, uma mudança de estado. O disparo de uma transição consiste em dois passos: (1) retirar as fichas dos lugares de entrada; e (2) colocar fichas nos lugares de saída.

Cada arco possui ainda um *peso* associado. O número de fichas a retirar dos lugares de entrada e o número de fichas a colocar nos lugares de saída por ocasião do disparo de uma transição é igual ao peso de cada arco correspondente[1]. A transição pode ocorrer, entretanto, somente se esta estiver *habilitada* (ou *sensibilizada*). Uma transição está habilitada se o número de fichas em cada um dos lugares de entrada for maior ou igual que o peso do arco que liga este lugar à transição.

Conforme Figura 2.2[2], os lugares, transições, fichas e arcos podem ser graficamente representados como um grafo orientado. A interpretação para a rede da Figura 2.2 é a seguinte: os lugares de entrada da transição "Iniciar operação" são "Máquina livre" e "Peça em espera". Visto que ambos os lugares de entrada da transição "Iniciar operação" possuem uma ficha cada na representação (a), diz-se que esta transição está habilitada. Isso significa que, nesse momento, o evento "Iniciar operação" pode ocorrer. Quando o evento ocorrer, são retiradas as fichas dos lugares "Máquina livre" e "Peça em espera", e é colocada uma ficha no lugar "Máquina em operação". A representação (b) da Figura 2.2 mostra o estado da rede de Petri depois do disparo de "Iniciar operação". Neste estado, o evento "Iniciar operação" não está mais habilitado e, portanto, não pode mais ocorrer.

[1] Por padrão, o peso de um arco é numericamente igual a um.

[2] Visto que essa figura não não inclui os pesos dos arcos, cada arco tem peso 1, por padrão.

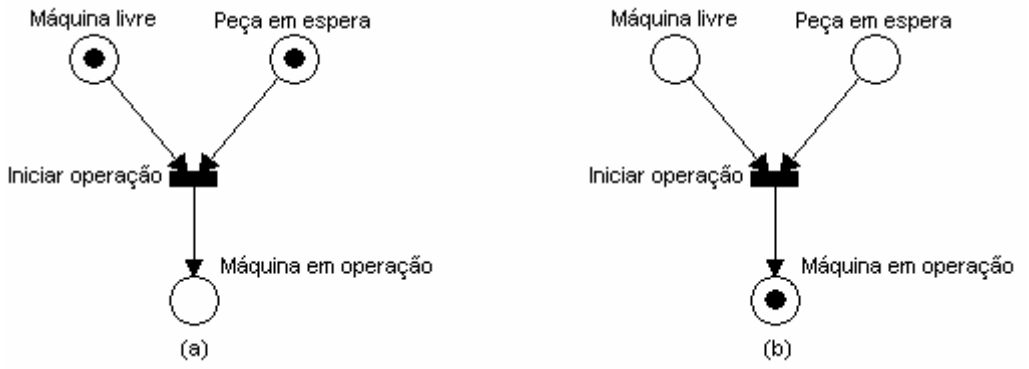


Figura 2.2: Exemplo de rede de Petri. Fonte: CARDOSO & VALETTE (1997)

2.2.2 Definição formal de rede de Petri

Segundo CARDOSO & VALETTE (1997), uma rede de Petri é uma quádrupla $R = (P, T, Pre, Post)$ onde:

a) P é um conjunto finito de lugares de dimensão n;
b) T é um conjunto finito de transições de dimensão m;
c) $Pre: P \times T \rightarrow IN$ é a aplicação de *entrada*, com IN sendo o conjunto dos números naturais;
d) $Post: P \times T \rightarrow IN$ é a aplicação de *saída*.

Uma rede de Petri não representa por si só o estado de um sistema. A evolução do estado de um sistema representado por uma rede de Petri é controlado por uma *marcação*. Uma rede de Petri marcada é uma dupla $N = (R, \mu)$ onde:

a) R é uma rede de Petri;
b) μ é a marcação inicial dada pela relação $\mu : P \rightarrow IN$, onde $\mu(p)$ é o número de fichas contidas no lugar p.

Como já dito anteriormente, uma rede de Petri marcada evolui o seu estado através do disparo de uma transição. Uma transição t pode ser disparada se, e somente se, esta estiver habilitada. Uma transição é dita habilitada se o número de fichas em cada um dos lugares de entrada for maior ou igual que o peso do arco que liga este lugar à transição, isto é, se e somente se $\forall p \in P, \mu(p) \geq Pre(p,t)$.

O disparo de uma transição t é uma operação que consiste em retirar $Pre(p,t)$ fichas de cada lugar precedente e colocar $Post(p,t)$ fichas em cada lugar seguinte. A

operação de disparo produz uma nova marcação para a rede de Petri, o que representa no modelo a mudança de estado do sistema devido à ocorrência do evento associado à transição *t*. Se uma transição qualquer *t* está habilitada por uma marcação μ, uma nova marcação μ' é obtida através do disparo de *t* tal que: $\forall p \in P, \mu'(p) = \mu(p) - Pre(p,t) + Post(p,t)$.

Como ilustrado na Figura 2.2, pode-se associar a uma rede de Petri um grafo com dois tipos de nós: nós lugares e nós transições. Os nós deste grafo são ligados com arcos regidos pelas seguintes regras:

a) um arco de peso *a* liga um lugar *p* a uma transição *t* se e somente se $a = Pre(p,t)$, com $a \neq 0$;
b) um arco de peso *b* liga uma transição *t* a um lugar *p* se e somente se $b = Post(p,t)$, com $b \neq 0$.

2.3 Programação Orientada a Aspectos

2.3.1 Motivação para o uso do paradigma de Programação Orientada por Aspectos

Segundo ELRAD et al. (2001), a AOP (*Aspect-Oriented Programming* – Programação Orientada a Aspectos) foi criada para resolver o problema da dispersão das áreas de interesse[1]. As áreas de interesse podem representar requisitos de alto nível, como segurança e qualidade de serviço, ou podem representar funcionalidades de baixo nível, como subsistemas de *caching* e *buffering*. O problema da dispersão de áreas existe na abordagem orientada a objetos pura, uma vez que o código que implementa uma área de interesse muitas vezes fica disperso em classes diferentes, localizadas horizontalmente na árvore de herança de um sistema de classes.

Ainda de acordo com ELRAD et al. (2001), a AOP é baseada na idéia de que os sistemas podem ser modelados em áreas de interesse e na descrição das relações entre estas áreas. Uma vez definidas as áreas, isto é, uma vez definidos os *aspectos*, cria-se um módulo que centraliza a implementação de uma área de interesse. Em

[1] O termo em inglês *concern* é traduzido neste trabalho como "área de interesse". ELRAD et al. (2001) cita os seguintes termos sinônimos de *concerns*: *properties* ou *areas of interest*.

seguida, utiliza-se um mecanismo que insere o código dos aspectos no código da aplicação, de forma que um programa coerente seja montado. A próxima seção mostra os principais conceitos da AOP aplicados na linguagem orientada a aspectos *AspectJ*.

2.3.2 A semântica das construções de *AspectJ*

Neste trabalho utiliza-se *AspectJ* (ECLIPSE, 2005), uma extensão de propósito geral da linguagem Java para a Orientação a Aspectos. Esta seção não pretende cobrir por completo uma revisão de todos os recursos que o *AspectJ* oferece, mas apenas os conceitos necessários para aplicação no controle de invocação de métodos em *frameworks* orientados a objetos.

Segundo KICZALES et al. (2001), as linguagens Orientadas a Aspectos possuem três elementos críticos: um modelo de *pontos de junção*[1]; os *pontos de corte*[2], uma forma de identificar os pontos de junção; e uma forma de modificar a implementação já compilada nos pontos de junção. Os pontos de junção provêem uma sintaxe de forma tal que é possível especificar a estrutura de propriedades relacionadas que estão dispersas em classes diferentes, localizadas horizontalmente na árvore de herança de um sistema de classes. Os seguintes pontos de junção são de interesse para aplicação neste trabalho:

a) *execução de método*: refere-se a quando o bloco de código que é corpo de um método concreto é executado;
b) *inicialização de objeto*: refere-se a quando o código de inicialização para uma classe específica é executado; compreende o tempo entre o início da chamada do primeiro construtor e o início da execução do construtor da superclasse.

A *AspectJ*, segundo XEROX (1998) ainda define, dentre outros, os seguintes *pontos de corte*:

a) *initialization(ConstructorPattern)* – especifica um ponto de junção particular de *inicialização de objeto* cuja assinatura é compatível com

[1] Do inglês *join points*.

[2] Do inglês *point cuts*.

ConstructorPattern;

b) *execution(MethodPattern)* – especifica um ponto de junção particular de *execução de método* cuja assinatura é compatível com *MethodPattern*;

c) *target(Type* ou *Id)* – especifica cada ponto de junção cujo objeto-alvo (o objeto que executa o método-alvo especificado por um ponto de corte, como *initialization* ou *execution*) é uma instância to tipo *Type* ou do tipo do identificador *Id*.

Outra construção de *AspectJ* relacionada aos pontos de corte são os *avisos*[1], o mecanismo utilizado para inserir código adicional que será executado nos pontos de corte. Em outras palavras, os avisos definem o comportamento das diferentes áreas de interesse que, de outra forma, estariam dispersas em classes diferentes, de maneira não-modularizada.

Em XEROX (1998) especifica-se um aviso da seguinte forma[2]:

```
[strictfp] <AdviceSpec> [throws <TypeList>] : <Pointcut> { <Body> }
```

onde <*AdviceSpec*> é uma das seguintes sentenças:

a) `before(<Formals>)`

b) `after(<Formals>) returning [ (<Formal>) ]`

c) `after(<Formals>) throwing [ (<Formal>) ]`

d) `after(<Formals>)`

e) `<Type> around(<Formals>)`

O tipo de aviso define o código que será executado em uma das situações:

a) *before* – antes do ponto de junção <*Pointcut*>;

b) *after* – depois do ponto de junção <*Pointcut*>;

c) *around* – o código que será executado no lugar do ponto de junção <*Pointcut*>.

---

[1] Do inglês *advice*.

[2] Nesta notação: os elementos entre colchetes são opcionais, os elementos entre "<" e ">" são não-terminais (derivam em outros elementos) e os demais são literais.

O código colocado em <*Body*> para o aviso *before* sempre é executado antes de um ponto de junção, porém *after* poderá ser executado ou não dependendo da forma como um ponto de junção termina: normalmente, indicado por *returning* [(<*Formal*>)]; depois de uma exceção ter sido lançada, indicado por *throwing* [(<*Formal*>)]; ou em qualquer uma das duas situações anteriores, indicado por *after*(<*Formals*>). O aviso *around* substitui o código original de seu ponto de corte pelo código colocado em <*Body*>.

O não-terminal <*Formals*> é utilizado para tornar disponível para o código <*Body*> um objeto que está sendo utilizado ou afetado pelo aspecto, ao passo que <*Formal*> indica um tipo simples (como *int* ou *bool*, por exemplo) ou uma classe.

2.3.3 Exemplificando na prática o uso de um aspecto

A fim de ilustrar a aplicação de um aspecto em um sistema real, considere o diagrama de classes da Figura 2.3. Nesse diagrama é apresentado o modelo de um editor de figuras. No editor de figuras, uma figura genérica (*Figure*) é um conjunto de Elementos (*FigureElement*). A classe *FigureElement* possui um método abstrato chamado *moveBy*, cuja responsabilidade é mover a figura dado um deslocamento horizontal e um deslocamento vertical. Cada subclasse de *FigureElement* define sua posição na figura de forma diferente. A classe *Point*, que representa um ponto, define sua posição com os métodos *setX* e *setY*, os quais determinam as coordenadas *x* e *y* do respectivo ponto. De maneira similar, a classe *Line*, que representa uma linha, define sua posição com os métodos *setP1* e *setP2*, os quais determinam os pontos de início e fim da mesma.

Os métodos *set* das classes *Point* e *Line* possuem uma característica em comum: cada vez que estes forem invocados, a classe *Display* (responsável pela apresentação dos elementos geométricos no monitor) deverá ser notificada disto e redesenhar a figura na tela. Essa notificação pode ser feita facilmente com a utilização de um aspecto, como mostra a Figura 2.4.

O aspecto da Figura 2.4, quando compilado com o código da aplicação, é usado para interceptar a execução dos métodos *set* e *moveBy* das classes *Point* e *Line*. Na prática, o aspecto *DisplayUpdating* invocará o método *update* da classe *Display* após a

execução destes métodos. Isto fará com que a tela seja atualizada quando os objetos do tipo *FigureElement* modificarem seus atributos.

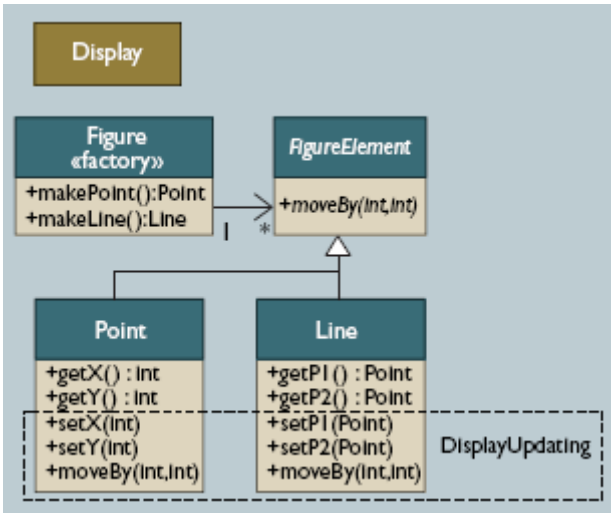


Figura 2.3: Exemplo de aplicação de um aspecto. Fonte: KICZALES et al. (2001)

```
aspect DisplayUpdating
{
    after(Point p):
        target(p) &&
        (execution(public void setX(int)) ||
         execution(public void setY(int))||
         execution(public void moveBy(int, int)))
    {
        Display.update(p);
    }

    after(Line line):
        target(line) &&
        (execution(public void setP1(Point)) ||
         execution(public void setP2(Point))||
         execution(public void moveBy(int, int)))
    {
        Display.update(line);
    }
}
```

Figura 2.4: Aspecto para atualizar a tela quando um ponto é deslocado

2.4 Conclusão

Um dos obstáculos ao uso de *frameworks* é a utilização correta dos objetos e classes abstratas que este provê. A utilização correta de um *framework* envolve a observância de suas restrições. Nesse contexto, as redes de Petri, como um modelo

geralmente usado para modelar sistemas discretos, podem ser usadas para validar algumas restrições que o projeto de um *framework* qualquer pode impor. As redes de Petri, entretanto, precisam de alguma forma ser ligadas ao código do *framework* para que a evolução do estado de suas marcações possa ser controlada. Considerando o disparo de transições de uma rede de Petri como uma área de interesse de uma aplicação, pode-se utilizar o paradigma da programação orientada a aspectos para alcançar este objetivo. A combinação desses três conceitos pode ser usada com sucesso para promover a usabilidade dos *frameworks* orientados a objeto, como será demonstrado no decorrer desse trabalho.

3 ESPECIFICAÇÃO DE RESTRIÇÕES DE PROJETO PARA *FRAMEWORKS* ORIENTADOS A OBJETOS E COMPONENTES

Muitas abordagens têm sido usadas pela comunidade científica para a especificação e validação de sistemas em geral. No contexto dos *frameworks* orientados a objetos e no desenvolvimento baseado em componentes reutilizáveis, são fornecidos a seguir os principais esforços direcionados na validação do uso de *frameworks* e componentes encontrados nos principais periódicos internacionais pesquisados nos últimos 5 anos. Embora este trabalho esteja direcionado especificamente aos *frameworks*, algumas abordagens de restrições ao uso de componentes serviram de base para a concepção deste trabalho. Por esta razão, incluiu-se também neste capítulo algumas referências ao desenvolvimento de componentes.

3.1 Especificação informal de *frameworks*

O método proposto por MURRAY et al. (2004) consiste em uma forma sistemática de documentar, via descrição textual informal, cada característica do *framework*. Essa descrição textual é criada como um meio de auxiliar o usuário do *framework* nas etapas a serem completadas a fim de produzir a aplicação final. Murray sugere a especificação das seguintes características de um *framework*: sintaxe, semântica, e a interface entre o *framework* e seus artefatos. Murray também declara que as etapas a serem completadas para instanciar de *framework* são: especificar as propriedades do sistema e prover os métodos concretos das classes abstratas. Convém destacar, entretanto, que as abordagens informais não conduzem a nenhum tipo de verificação automática; além disso, descrições informais podem ser dúbias e passíveis de serem mal interpretadas.

3.2 Validação do uso de *frameworks* através de linguagens de domínio e linguagens de restrições

FONTOURA et al. (2000) sugere a criação de uma DSL (*Domain Specification*

Language – Linguagem de Especificação de Domínio) específica para cada *framework* orientado a objetos. Com o uso de uma DSL, Fontoura sugere a criação de um compilador que aceita como entrada uma especificação escrita em linguagem DSL e o código da aplicação que usa o *framework*. Antes de executar a aplicação, esse compilador analisa o código e gera mensagens apropriadas se houver qualquer violação das restrições ao uso do *framework*. Fontoura apresenta dois estudos de caso para demonstrar sua abordagem. No primeiro estudo de caso, uma DSL é definida para especificar quais são os "pontos de variação" (*variation points*) de um *framework* de heurísticas de procura, e quais dos pontos de variação o usuário deste *framework* obrigatoriamente deve implementar para usá-lo corretamente. A DSL criada para este caso permite também a criação automática de *stubs* para os métodos que o usuário do *framework* deve implementar, além de possibilitar a verificação de certos predicados associados à lógica do próprio *framework*. O segundo estudo de caso trata-se de um *framework* para supermercados virtuais (uma aplicação de *e-commerce*). Este *framework* foi projetado para ser flexível à inclusão de novos produtos diferentes no catálogo de produtos do supermercado. Para este *framework*, utiliza-se uma DSL em formato XML para validar o *framework* que é extendido para suportar um novo produto.

HOU & HOOVER (2001) propõem uma linguagem para a especificação de restrições ao uso de *frameworks* em geral chamada FCL (*Framework Constraints Language* – Linguagem para Restrições de *Frameworks*). A FCL é baseada na lógica de predicados de primeira ordem e na teoria dos conjuntos, embora sua sintaxe seja formada por elementos que se aproximam o máximo possível das linguagens de programação. Assim como o uso de DSL (FONTOURA et al., 2000), a FCL analisa o código da aplicação final construída sob o *framework* de maneira estática: se houver qualquer violação, mensagens de erro são geradas antes da execução da aplicação. HOU & HOOVER (2001) ilustram sua abordagem com dois *frameworks:* um para sistemas bancários e o MFC (*Microsoft Foundation Classes*), onde a FCL é usada para declarar explicitamente as asserções que precisam ser observadas pelo código que instancia estes *frameworks*.

A abordagem de Hou difere da abordagem de Fontoura no sentido de que a FCL é aplicável a qualquer *framework*. Por outro lado, o método de Fontoura sugere que seja criada uma linguagem de descrição específica para cada *framework*.

3.3 Abordagens baseadas em processo de desenvolvimento de *software*

OLIVEIRA et al. (2004) propõem uma abordagem de instanciação de *frameworks* que é baseada em um processo de desenvolvimento de *software*. Oliveira argumenta que o uso de um processo sistemático e específico ao caso do desenvolvimento de aplicações construídas sob *frameworks* pode conduzir à produção de código que respeite as restrições que um *framework* específico impõe. Essa abordagem alcança estes objetivos por aplicar os seguintes métodos: especificação do *framework* através de uma extensão da UML chamada UML-FI (UML-*Framework Instantiation*), e o uso de uma linguagem de especificação de reuso baseada em processo de desenvolvimento de *software*. A combinação desses dois métodos indica "como" a instanciação deve ser feita, "quais" atividades precisam ser completadas, e "quando" cada atividade deve ser escalada para que as restrições do *framework* possam ser observadas. Esta proposta toma como entrada, dentre outros, um documento chamado Conjunto de Restrições. O Conjunto de Restrições é usado para descrever as restrições estruturais que podem ser verificadas por uma ferramenta que avalia a integridade do sistema como um todo. As restrições estabelecidas pelo método sugerido por Oliveira tratam especificamente daquelas relacionadas com as propriedades estruturais dos modelos instanciados, como anomalias estruturais, aderência a princípios de projeto, e completude. Oliveira declara explicitamente que sua abordagem não trata a questão comportamental da instanciação de *frameworks,* visto que seu método de verificação utiliza apenas técnicas de análise estática baseadas em XML (W3C, 1996) e XSLT (CLARK, 2003).

SILVA & FREIBERGER (2004) apresentam um conjunto de métricas extraídas do desenvolvimento de um conjunto de aplicações construídas sob um *framework* em particular. Esse conjunto de métricas é utilizado para avaliar se a aplicação em desenvolvimento está de acordo com as métricas das aplicações desenvolvidas anteriormente. A análise qualitativa destas informações, consolidadas em um banco de dados, é utilizada para informar ao desenvolvedor se há algo no seu código que foge ao padrão utilizado pela maioria das aplicações desenvolvidas até o momento. Essa análise potencialmente ajudaria o desenvolvedor a avaliar uma eventual quebra das restrições estabelecidas ao uso do *framework* que está sendo extendido.

As abordagens de OLIVEIRA et al. (2004) e SILVA & FREIBERGER (2004)

destacam a importância de um processo bem definido para a criação de aplicações que utilizam *frameworks* orientados a objetos. Oliveira define explicitamente um processo, enquanto Silva define como fazer a extração de métricas de forma automatizada e o armazenamento destas informações em um banco de dados de histórico de aplicações. A inclusão da análise de métricas de Silva acrescentaria no processo de Oliveira uma importante característica de processos de desenvolvimento maduros: o uso de métricas e histórico de desenvolvimento para a concepção de novos sistemas.

3.4 Validação e especificação de sistemas orientados a objetos com redes de Petri

THANH & KLAUDEL (2004) propõem uma extensão para uma linguagem de especificação, $B(PN)^2$, a qual foi criada para expressar os conceitos tradicionais da computação paralela, como composição paralela, iteração, procedimentos e comunicações. A linguagem $B(PN)^2$ possui também uma semântica expressa em termos de redes de Petri coloridas, chamadas *M-nets*. Essa característica da linguagem $B(PN)^2$ foi usada em THANH & KLAUDEL (2004) para a criação da BOON (*Basic Object-Oriented Notation* – Notação Básica Orientada a Objetos), uma notação cujo objetivo é a representação de conceitos da Programação Orientada a Objetos, como classes, métodos, atributos, herança, polimorfismo e ligação dinâmica (*dynamic binding*) aplicados aos sistemas paralelos. As *M-nets* são usadas também pela BOON para modelar eventos como criação e destruição de objetos e chamada de métodos. Alguns princípios que também fundamentaram o projeto da BOON foram a verificação automatizada dos conceitos representados, bem como a criação automática de diagramas UML.

SILVA & PRICE (2002) utilizam redes de Petri ordinárias para a especificação de interface de componentes[1]. Um padrão de projeto é proposto para modelar a compatibilidade comportamental e estrutural da composição de componentes genéricos. Modelam-se os elementos dos componentes com redes de Petri, e qualquer inconsistência na rede de Petri resultante da composição de componentes (como *deadlocks*, por exemplo) indicará que os componentes são incompatíveis.

[1] O desenvolvimento baseado em componentes é amplamente abordado em SILVA (2000).

As duas abordagens, de THANH & KLAUDEL (2004) e SILVA & PRICE (2002), modelam conceitos em diferentes níveis de abstração; Thanh se atém aos elementos de baixo nível da programação orientada a objetos, enquanto que Silva direciona o trabalho em componentes genéricos (não necessariamente orientado a objetos) de nível de abstração mais altos. Os dois trabalhos, entretanto, possuem uma característica comum: a validação de sistemas com o uso de redes de Petri; ambos desejam verificar o sistema final.

3.5 Especificação formal com técnicas baseadas em Programação Orientada a Aspectos

UBAYASHI & TETSUO (2002) propõem uma abordagem baseada em verificação de modelo para verificar se um sistema compilado com a introdução de aspectos possui comportamento adequado. Ubayashi também apresenta um *framework* orientado a aspectos projetado para verificar as asserções de outros *frameworks*.

O *framework* de verificação de modelo proposto em UBAYASHI & TETSUO (2002) tem especial relação com este trabalho de pesquisa. Ubayashi utiliza aspectos para verificar se as pré e pós-condições de métodos de um sistema são observados. A Figura 3.1 mostra de forma esquemática como funciona esse *framework* de verificação.

Em UBAYASHI & TETSUO (2002), as verificações introduzidas no código através dos aspectos são feitas através do Jpf (Java *PathFinder*) (NASA, 2005). O Jpf permite a escrita de asserções que devem ser observadas no ponto onde a asserção é declarada. Para fazer a verificação destas asserções, o Jpf toma como entrada o código Java final compilado (os arquivos em formato *bytecode*) e traça uma árvore com todos os caminhos possíveis que o fluxo de execução pode tomar. Se, em algum desses caminhos, qualquer asserção for violada, o usuário será notificado disso. As asserções utilizadas pelo Jpf são escritas na própria linguagem Java e ficam embutidas no próprio código da aplicação final. O diferencial introduzido pelo *framework* proposto em UBAYASHI & TETSUO (2002) é a retirada das asserções que ficam embutidas no código da aplicação e a introdução dessas asserções no código compilado via aspectos.

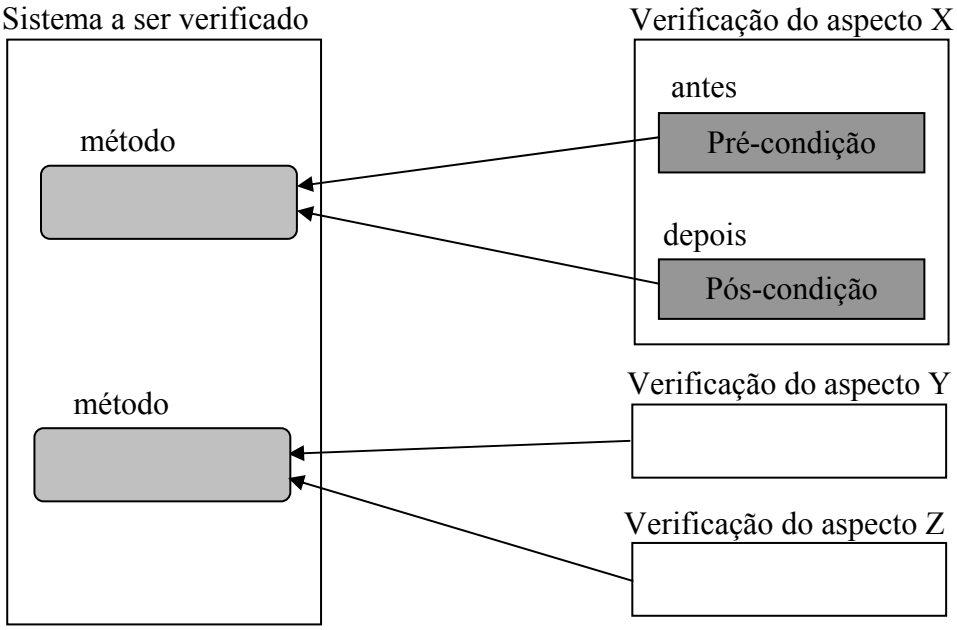


Figura 3.1: *Framework* de verificação de modelo baseado em AOP. Fonte: adaptado de UBAYASHI & TETSUO (2002)

Outra abordagem, publicada por LAM et al. (2005) utiliza conceitos de AOP para especificar três construções: formatos, escopos e *defaults*. Essas construções, em combinação com uma linguagem de especificação baseada em conjuntos abstratos de objetos, permitem a verificação da consistência de propriedades estruturais em sistemas extensos. LAM et al. (2005) argumenta que as abordagens tradicionais de especificação de pré e pós-condições são muitas vezes replicadas e dispersas nas diferentes unidades de código que constituem uma aplicação. Esta replicação é denominada de *agregação de especificação*. A agregação de especificação, em termos simples, trata-se do fato de que as pré-condições de um procedimento precisam, obrigatoriamente, incluir todas as pré-condições dos procedimentos que este invoca. Nesse sentido, a abordagem de Lam propõe a eliminação da agregação de especificação com as construções: formatos, escopos e *defaults.* A proposta de Lam, entretanto, é direcionada principalmente para a verificar a consistência das estruturas de dados utilizadas pelo sistema.

Uma característica dos modelos de UBAYASHI & TETSUO (2002) e LAM et al. (2005) é a escrita de especificação formal de baixo nível, especificação esta que envolve a compreensão de estruturas complexas e aspectos. Com relação à verificação das asserções, o método de Lam utiliza análise estática, enquanto que a utilização de Jpf

por Ubayashi possibilita a verificação a partir da análise exploratória dos estados do programa final compilado.

3.6 Análise do estado da arte na especificação de restrições ao uso de *frameworks*

Com exceção de MURRAY et al. (2004), todas as abordagens pesquisadas tratam o problema da especificação de sistemas orientados a objetos e ao uso de *frameworks* de maneira formal. Nota-se claramente a tendência geral pela especificação formal em detrimento dos métodos textuais informais. Todos os artigos que tratam de especificação formal de restrições aplicam os métodos propostos em *frameworks* do mundo real, o que mostra que o formalismo é factível de uso na prática. Ainda com relação aos métodos formais, MURRAY et al. (2004), declaram explicitamente que uma extensão ao seu trabalho seria a criação de uma linguagem formal para especificar a sintaxe e a semântica de *frameworks* e seus artefatos.

As abordagens formais mencionadas neste capítulo, com exceção de THANH & KLAUDEL (2004) são técnicas de análise estática, isto é, a verificação das asserções é feita baseada apenas no código-fonte da aplicação que foi previamente escrito, e *antes* da execução do sistema. A análise estática possibilita, por exemplo, a verificação de visibilidade de métodos e a verificação da implementação dos *hot spots* (HOU & HOOVER, 2001). Por outro lado, a verificação estática precisa avaliar todos os caminhos de execução a que o sistema pode ser conduzido, e este processo pode ser dispendioso, mas necessário, para a avaliação das restrições do *framework*. Para resolver esse problema, HOU & HOOVER (2001) incluem em sua análise estática, por exemplo, o tratamento de estruturas de repetição (*while, for,* etc) e o tratamento de estruturas de decisão (*if, case,* etc). Devido a essa característica, o método de HOU & HOOVER (2001) é dependente de linguagem; se forem incluídas novas construções na linguagem-alvo na qual o *framework* é implementado, isso afetará a descrição formal e a verificação da conformidade do sistema com a especificação. Nesse sentido, UBAYASHI & TETSUO (2002) propõem o uso de aspectos para a inclusão de verificações de Jpf (NASA, 1999), o qual se encarrega de avaliar os possíveis caminhos de estados em que o sistema pode tomar, tendo como entrada o código da aplicação já compilado. O Jpf, entretanto, possui as seguintes limitações (NASA, 1999): não faz a

verificação de programas que dependem de métodos nativos (código dependente de plataforma); não suporta programas que utilizam classes dos pacotes *java.awt* (classes que implementam interface gráfica com o usuário), *java.net* (classes para trabalhar com sockets e funcionalidades relacionadas à Internet), e suporta parcialmente classes do pacote *java.io* (classes que tratam dispositivos de entrada e saída, como arquivos, por exemplo); e não é capaz de verificar programas com tamanho superior a 10000 linhas de código (devido a requisitos de armazenagem de estados).

Dadas as limitações da abordagem de UBAYASHI & TETSUO (2002) e de Jpf (NASA, 1999), este trabalho propõe um método que utiliza análise dinâmica (verificação do sistema enquanto este está sendo testado e realmente executado) em conjunto com a execução dos testes funcionais do sistema final. Durante a fase de testes, podem ser geradas mensagens de erro caso alguma violação do *framework* tenha ocorrido. A verificação dinâmica em conjunto com os testes funcionais do sistema final supera as limitações de Jpf (NASA, 1999) e não preocupa-se com as estruturas de repetição e decisão de uma linguagem específica (HOU & HOOVER, 2001). Além disso, as técnicas de análise estáticas não conseguem simular completamente o ambiente em que o sistema será executado. Logo, os testes funcionais dificilmente serão abandonados para serem substituídos pela análise estática somente.

Com relação ao nível de abstração da especificação formal proposta pelos métodos mencionados neste capítulo, observa-se uma predominância do uso de especificações no nível de estruturas de dados e variáveis locais. Uma das formas pesquisadas de especificar restrições é utilizando a própria linguagem em que o *framework* foi escrito, como, por exemplo, algo do tipo: “neste ponto, a variável *x* não pode ser nula” (UBAYASHI & TETSUO, 2002). Outra forma de especificar restrições é pelo uso de lógica de primeira ordem e teoria dos conjuntos. De acordo com essa abordagem, não se observam restrições estabelecidas em nível mais alto de abstração, como, por exemplo, “o método *y* tem que ser invocado antes de *z,* caso contrário a variável *x* será nula e isto produz um erro na aplicação final”. Neste trabalho, optou-se por utilizar uma especificação formal em alto nível para resolver especificamente o problema da ordem de invocação de métodos. Embora as abordagens de especificação em baixo nível possam estabelecer uma classe de restrições bem mais abrangente, considera-se que o tempo para produzir especificações em baixo nível é alto demais, e,

além disso, a manutenção desse formalismo seria impraticável para muitos casos.

Com relação ao formato da especificação formal, THANH & KLAUDEL (2004) utilizam a notação BOON para modelar características de modelos orientados a objeto em geral, a qual demanda conhecimento avançado dessa notação e de $B(PN)^2$ (a notação da qual BOON é derivada, projetada para sistemas concorrentes), forçando o desenvolvedor do *framework* a produzir descrições praticamente ilegíveis e difíceis de entender. HOU & HOOVER (2001) utilizam FCL, uma linguagem baseada em lógica de primeira ordem e teoria dos conjuntos. OLIVEIRA et al. (2004) especifica as restrições ao uso de *frameworks* com XML e XSLT, mas não torna claro como isso é realmente feito. O método de SILVA & PRICE (2002) exige que o desenvolvedor modele os componentes com redes de Petri, o que pode ser um desafio se o desenvolvedor não tiver experiência com este modelo. No geral, a forma de especificar as restrições de *frameworks* ou componentes demandam conhecimento de algum método formal de baixo nível. Utiliza-se neste trabalho, portanto, notação XML para especificar apenas a ordem de restrição ao uso de *frameworks*, dada a ampla disseminação dessa notação pela Internet. Mais uma vez, reconhece-se que a expressividade da classe de restrições que esse método pode representar é limitada, mas visto que na maioria das vezes o desenvolvedor dispõe de pouco tempo para desenvolver o seu sistema, a especificação via métodos com notações de baixo nível pode ser proibitiva em termos do tempo que essas consomem. A abordagem utilizada neste trabalho utiliza redes de Petri como meio de controlar o estado do sistema, mas isso é completamente transparente, tanto para o desenvolvedor quanto para o usuário do *framework*. Essa característica deverá motivar ambos os desenvolvedores a fazerem uso do método sugerido.

Em suma, a análise do estado da arte em especificação de sistemas aplicada aos *frameworks* orientados a objetos levaram aos seguintes princípios orientadores da presente pesquisa:

a) o uso de especificação formal em detrimento aos métodos não formais;
b) a verificação da conformidade do sistema com a especificação formal deve acontecer em conjunto com os testes funcionais do sistema, enquanto o *software* estiver sendo realmente executado; isto demanda

um bom planejamento de testes, e estes devem ser feitos de forma sistemática, destacando a importância de um processo de desenvolvimento bem definido;

c) a especificação do *framework* deve ser escrita em um alto nível de abstração, independente da sua linguagem alvo e sem utilizar notações pouco conhecidas pelos desenvolvedores de *software* do meio corporativo;
d) nem o desenvolvedor nem o usuário do *framework* deverão precisar conhecer modelos formais como redes de Petri ou $B(PN)^2$.

4 ESPECIFICAÇÃO FORMAL DE RESTRIÇÕES DE PROJETO PARA *FRAMEWORKS* E CONTROLE DE ESTADO

Conforme declarado no capítulo 1, o objetivo deste trabalho é garantir que um *framework* seja utilizado de forma correta. Nesse sentido, o foco adotado é o de estabelecer restrições formais à ordem de invocação de métodos providos pelos objetos do *framework*, isto é, garantir que a seqüência em que os métodos são chamados adequa-se a um "contrato" estabelecido pelo projetista do *framework*.

4.1 O processo de validação formal da ordem de chamada de métodos de *frameworks*

O uso de um *framework* para a criação do *software* final deverá ser validado formalmente seguindo-se os passos ilustrados na Figura 4.1.

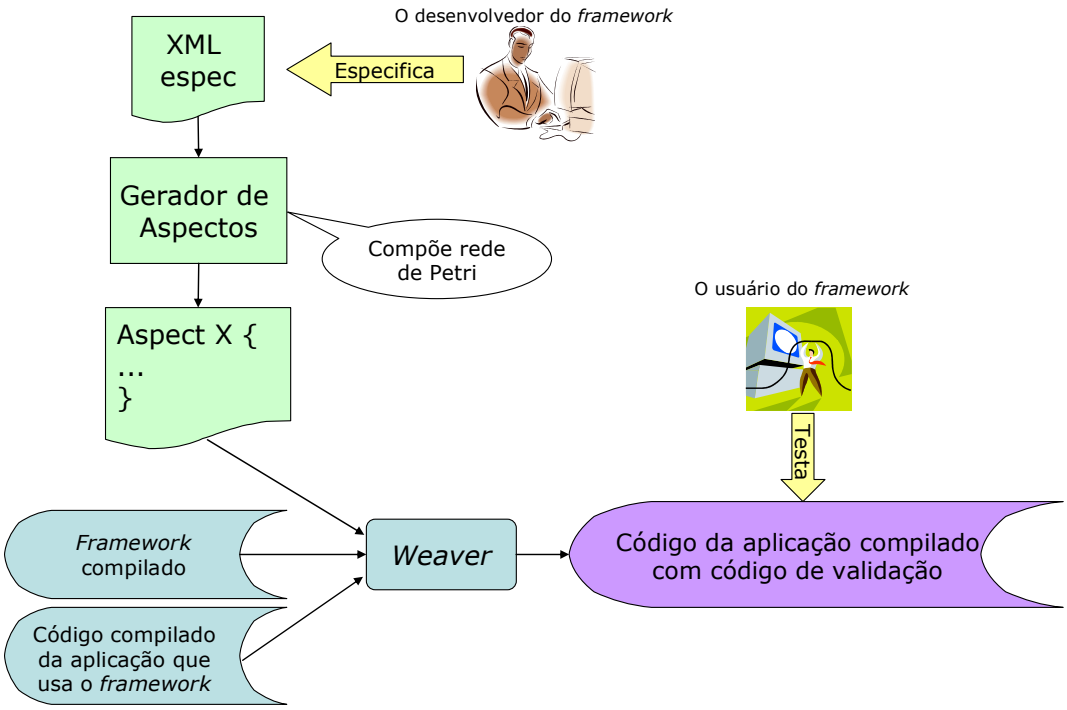


Figura 4.1: O processo de validação formal da ordem de chamada de métodos de *frameworks*

O processo de validação da ordem de chamada de métodos exige que o desenvolvedor do *framework* especifique formalmente a ordem em que os métodos do mesmo podem ser invocados. Esta especificação ("XML Espec") é escrita em formato XML – *eXtensible Markup Language* (W3C, 1996), e servirá de entrada para a ferramenta "Gerador de Aspectos". O Gerador de Aspectos, de forma transparente tanto para o desenvolvedor quanto para o usuário do *framework,* utilizará a especificação formal para compor uma Rede de Petri, a qual será responsável pelo controle da invocação de métodos. O aspecto automaticamente gerado servirá de entrada para o *Weaver*. O *Weaver* também tomará como entrada o código compilado do *framework* e o código compilado da aplicação que usa este *framework*, gerando uma versão do *software* que deverá ser testada. Durante a fase de testes, o aspecto criado pelo Gerador de Aspectos indicará ao testador qualquer chamada indevida aos métodos do *framework*.

Esse processo de validação não implica em esforço adicional por parte do desenvolvedor que utiliza o *framework* para criar o *software* final. Por outro lado, é requerido do desenvolvedor do *framework* a especificação formal deste. Embora a especificação demande tempo, o esforço empenhado na especificação é economizado posteriormente na fase de testes das aplicações que utilizam o *framework*.

4.2 Máquinas de estado e controle de invocação de métodos

A fim de controlar a chamada de métodos de um objeto para evitar o seu mau uso, é preciso saber, em um determinado momento, quais métodos deste objeto já foram invocados e quais podem ser chamados a partir de então. A relação entre o estágio de evolução de um objeto e os métodos a invocar pode ser estabelecida e controlada através de uma *máquina de estados finitos*. Controla-se o *estado* de um objeto em particular associando-se a ele uma máquina de estados finitos.

Segundo CARDOSO & VALETTE (1997), uma máquina de estados finitos é a representação de um sistema orientado a eventos discretos, e é definida como uma quádrupla $Me = (K, s_0, \Sigma, \delta)$ onde:

a) K é um conjunto finito não-vazio de *estados*;

b) s_0 é o estado inicial, onde $s_0 \in K$;

c) Σ é o alfabeto, finito, de entrada[1];

d) $\Delta: K \times \Sigma \rightarrow K$ é a função de mapeamento.

No caso de uma classe em particular, define-se para ela uma máquina de estados da seguinte forma:

a) cada estado, exceto o estado inicial, representa o momento imediatamente posterior ao término da execução de um método específico, respeitando a ordem correta de chamada;

b) o estado inicial representa o momento imediatamente posterior à criação do objeto, isto é, o momento em que o construtor da classe termina sua execução;

c) o "alfabeto" de entrada representa o conjunto de métodos da classe em questão;

d) a função de transição especifica a dependência entre a invocação dos métodos de uma classe (isto é, a ordem de chamada);

A proposta deste trabalho consiste em controlar o comportamento *dinâmico* do sistema. Isto significa que a máquina de estados acima definida será criada em tempo de *execução* e será transicionada no momento real em que os métodos são chamados. Isso difere substancialmente do método proposto por HOU & HOOVER (2001) e OLIVEIRA et al. (2004), onde são definidas linguagens de especificação de *frameworks* baseadas em técnicas de análise estáticas. Nenhuma das duas abordagens trata a questão *dinâmica* do uso do *framework*.

Ainda neste capítulo, é explanada a forma pela qual a máquina de estados que verificará a ordem de invocação de métodos pode ser instanciada em tempo de execução, e como um módulo para isto é implementado de maneira que o código das classes do *framework* e o código das classes da aplicação não sejam alterados. Esse módulo de verificação é gerado automaticamente a partir da especificação formal, liberando o usuário do *framework* da tarefa de fazer esta verificação manualmente.

---

[1] O alfabeto é um conjunto finito de eventos discretos que disparam as transições de estado do sistema.

4.2.1 Exemplificando o uso de máquinas de estado para restringir o uso de um *framework*

Para exemplificar uma situação de como uma máquina de estados conforme o modelo da seção anterior pode ser ser definida, considere o *framework*-exemplo extraído de HOU & HOOVER (2001) ilustrado na Figura 4.2[1]. O *framework* da Figura 4.2 é o modelo simplificado de um sistema bancário. As classes que pertencem ao *framework* são *ATM*[2] e *Account*, ao passo que *BillingATM* foi criada para produzir a aplicação completa.

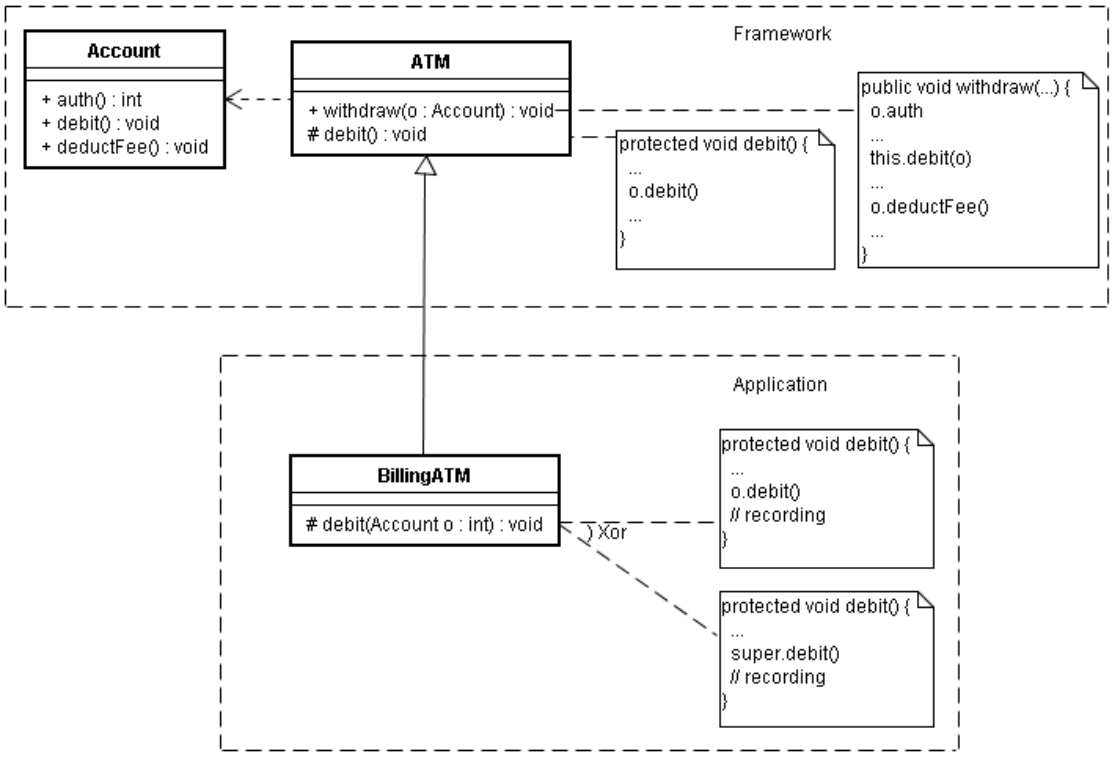


Figura 4.2: Um exemplo de restrição de uso de um *framework*. Fonte: HOU & HOOVER (2001)

A classe *Account* implementa três métodos: *auth* para autenticar um cliente; *debit* para debitar uma quantia em dinheiro da conta do cliente; e *deductFee* para

[1] A figura original contida em HOU & HOOVER (2001) apresenta código escrito em C++. Neste trabalho, optou-se por utilizar a linguagem Java; o código escrito nesta figura foi adaptado segundo esta linguagem. O diagrama desta figura está escrito em notação UML (OMG, 1997).

[2] A sigla ATM vem do inglês *Automatic Teller Machine*, usada para referir-se a caixas eletrônicos de bancos usados para sacar dinheiro, pagar contas, emitir talões de cheque, etc.

calcular as taxas a serem cobradas pela transação. Esta classe admite que seus métodos sejam invocados na seguinte ordem:

1. *auth*
2. *debit*
3. *deductFee*

A classe ATM implementa dois métodos: *withdraw* e *debit*. O método *withdraw* é um método *template*[1] (GAMMA et al., 1994) que invoca *auth* (de *Account*), o método *debit* (que é um método *hook*) e *deductFee* (de *Account*) (nesta ordem). O método *debit* de *ATM* poderá ser sobrescrito por uma subclasse de *ATM*. De qualquer forma, sua implementação padrão invoca o método *debit* da classe *Account* somente uma vez.

A classe *BillingATM* é derivada da classe *ATM* do *framework*, cujos serviços são restritos a somente pagar uma conta. *BillingATM* acrescenta alguma funcionalidade de registro adicional ao método *debit* além do comportamento padrão de sua superclasse. Ao reimplementar este método, o usuário do *framework* poderá invocar o método *debit* de *Account* somente uma vez. Caso contrário, a restrição de chamada de métodos da classe *Account* não é atendida.

Para formalizar a restrição que há na ordem de chamada dos métodos da classe *Account* e evitar que o usuário do *framework* a utilize de forma indevida, propõe-se a seguinte máquina de estados:

a) o conjunto de estados é
   K = {OBJECT_CREATED, AUTH_FINISHED,
   DEBIT_FINISHED, DEDUCTFEE_FINISHED};
b) o estado inicial é
   s_o = OBJECT_CREATED;
c) o alfabeto de entrada é
   Σ = {*auth*, *debit*, *deductFee*};
d) a função de mapeamento é
   Δ = {((OBJECT_CREATED, *auth*), AUTH_FINISHED),
   ((AUTH_FINISHED, *debit*), DEBIT_FINISHED),

[1] Segundo GAMMA et al. (1994), o padrão de projeto *template* define um algoritmo fixo em função de métodos abstratos, chamados de *hook*, os quais serão implementados na sub classe concreta.

((DEBIT_FINISHED, *deductFee*), DEDUCTFEE_FINISHED)};

A máquina de estados definida acima pode ser representada graficamente pelo diagrama de transição de estados da Figura 4.3[1].

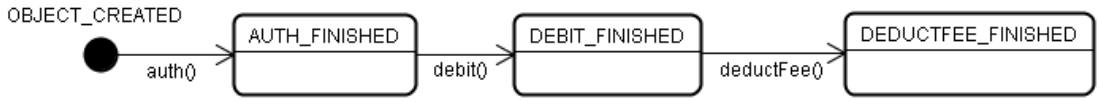


Figura 4.3: Diagrama de transição de estados para a classe *Account*

4.3 Especificação formal de interface de classes e geração automática de máquinas de estado

Conforme explanado na seção anterior, a ordem de invocação dos métodos de uma classe pode ser controlada com uma máquina de estados finitos. Segundo AHO et al. (1986), um AFN (Autômato Finito Não-determinístico) é um reconhecedor[2] de conjuntos regulares[3]. A definição de Aho para um AFN é semelhante à definição de uma máquina de estados finitos conforme definido na seção 4.2; um AFN acrescenta à definição de máquina de estados finitos um conjunto de estados finais[4]. Apesar disso, um AFN pode ser considerado equivalente a uma máquina de estados finitos no contexto deste trabalho, visto que o conjunto de estados finais não representa nada significativo à ordem de invocação dos métodos de uma classe. Ainda segundo AHO et al. (1986), para todo conjunto regular existe uma *expressão regular* que o denota inequivocamente. Portanto, se um AFN é equivalente a uma máquina de estados finitos,

---

[1] O diagrama desta figura foi escrito em notação UML (OMG, 1997).

[2] Segundo AHO et al. (1986), um *reconhecedor* para uma linguagem é "um programa que toma como entrada uma cadeia *x* e responde 'sim' se *x* for uma sentença da linguagem e 'não' em caso contrário".

[3] Segundo AHO et al. (1986), um conjunto regular é a linguagem gerada por uma *gramática regular* e é gerado pelas seguintes operações sob dois conjuntos finitos: união, concatenação, fechamento de Kleene e fechamento positivo.

[4] Os AFNs são usados geralmente no contexto das linguagens formais aplicadas aos compiladores. Os estados finais são usados para indicar os estados em que um AFN reconhece uma sentença como pertencente a uma linguagem.

se um AFN reconhece um conjunto regular e se uma expressão regular denota corretamente este conjunto, então a utilização de uma expressão regular torna-se apropriada para especificar a ordem em que os métodos de uma classe podem ser invocados.

AHO et al. (1986) ainda define as regras que regem as expressões regulares sobre um alfabeto[1] Σ da seguinte forma:

> *"1.* ε*, a cadeia vazia, é uma expressão regular que denota* {ε}*, isto é, o conjunto que contém a cadeia vazia.*
> *2. Se* a *é um símbolo em* Σ*, então* a *é uma expressão regular que denota* {a}*,isto é, o conjunto contendo a cadeia* a*.*
> *3. Se* r *e* s *são expressões regulares denotando as linguagens* L(r) *e* L(s)*, então:*
> *a)* (r)|(s) *é uma expressão regular denotando* L(r) ∪ L(s).
> *b)* (r)(s) *é uma expressão regular denotando* L(r) L(s).
> *c)* (r)* *é uma expressão regular denotando* (L(r))*.
> *d)* (r) *é uma expressão regular denotando* L(r)*." (o grifo é nosso)*

Na definição acima, a barra vertical significa "ou", os parênteses são usados para agrupar subexpressões, o asterisco significa "zero ou mais instâncias" da expressão entre parênteses e a justaposição (*r*)(*s*) significa concatenação. Parênteses desnecessários também podem ser evitados seguindo a seguinte convenção de ordem de precedência (da esquerda para a direita): (1) operador "*"; (2) concatenação; e (3) operador "|". Na escrita de uma expressão regular, também é comum a utilização das seguintes abreviações para efeitos de simplificação: (1) $p^+ = pp^*$; e (2) $p^? = p \mid \varepsilon$.

Conforme já mencionado, uma expressão regular é apropriada para descrever formalmente a ordem de invocação dos métodos de uma classe. Observando-se as regras que regem uma expressão regular, especifica-se a interface de uma classe da seguinte forma:

a) o "alfabeto" de entrada é o conjunto composto pelos métodos da classe que está sendo especificada;
b) se *r* e *s* são dois métodos de uma classe, então *r*|*s* denota que ou *r* ou *s* podem ser invocados;
c) se *r* e *s* são dois métodos de uma classe, então *rs* significa que *r* deve

[1] O termo *alfabeto* no contexto dos AFNs representa um conjunto finito de símbolos, como letras ou dígitos, por exemplo.

ser invocado primeiro, e na seqüencia *s* deve ser invocado;

d) se *r* é o método de uma classe, então $r^?$ significa que a invocação de *r* é opcional;

e) se *r* é o método de uma classe, então r^+ significa que *r* deve ser invocado pelo menos uma vez.

4.3.1 Exemplificando a especificação de uma classe com arquivos XML

Neste trabalho, adotou-se o formato dos arquivos XML – *eXtensible Markup Language* (W3C, 1996) para a escrita da expressão regular que descreve formalmente a interface de uma classe específica. O uso de XML para produzir a especificação foi precedido pelos trabalhos de FONTOURA et al. (2000) e OLIVEIRA et al. (2004). A estrutura do arquivo XML que especifica a ordem de chamada dos métodos de uma classe é por sua vez especificada com um *esquema* XML. Os esquemas XML (W3C, 2000) são um mecanismo de descrição formal de documentos XML e que são usados neste trabalho como ferramenta para a especificação de interface de classes. Deste modo, a classe é especificada segundo os padrões de *tags* de documentos XML seguindo o esquema provido (esse esquema está listado no Anexo 1). O esquema admite uma entrada que equivale a uma linguagem regular, a qual é convertida em uma máquina de estados. Na próxima seção são explicados os motivos pelos quais esta conversão é necessária e como esta pode ser realizada.

A fim de exemplificar como o esquema XML ora proposto funciona, reportemo-nos novamente ao *framework*-exemplo da Figura 4.2. Neste *framework*, a classe *Account* admite que os métodos *auth*, *debit* e *deductFee* sejam invocados nesta ordem. Assim, seguindo o esquema do Anexo 1, especifica-se sua interface conforme o arquivo XML mostrado na Figura 4.4.

```
<?xml version="1.0" encoding="UTF-8"?>
<systemSpecification
    xmlns="http://www.motorola.com/systemspecification"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.motorola.com/systemspecification
D:\users\rechia\rechia_mestrado\taf_petri10\system_validation\src\java
\com\motorola\systemvalidation\schemas\system_schema.xsd"
    xmlns:xlink="http://www.w3.org/1999/xlink" xlink:type="extended">
```

(Figura 4.4: continua na próxima página)

(Figura 4.4: continuação da página anterior)

```
    <class name="Account">
      <methodsCallOrder>
         <list type="sequence">
             <methodCall signature="public int auth()"/>
             <methodCall signature="public void debit()"/>
             <methodCall signature="public void deductFee()"/>
         </list>
      </methodsCallOrder>
    </class>
</systemSpecification>
```

Figura 4.4: Especificação formal da classe *Account*

Na especificação da Figura 4.4, os atributos do elemento *systemSpecification* definem o *namespace* (W3C, 1999) e a localização do esquema do arquivo XML. Em seguida, o atributo *name* do elemento *class* define a classe que será especificada. O elemento *class*, por sua vez, possui um sub-elemento chamado *methodsCallOrder* responsável por declarar a ordem de invocação dos métodos desta classe. Ainda interno à *methodsCallOrder*, define-se um elemento do tipo *list* o qual possui um atributo chamado *type*. Neste caso, o atributo *type* é definido como *sequence*, indicando que os métodos declarados como sub-elementos de *list* terão que ser executados obrigatoriamente em seqüência. Uma *list* do tipo *sequence* é equivalente ao operador de concatenação das expressões regulares.

A definição dos sub-elementos de *list* é recursiva (ver o Anexo 1). Isto significa que *list* é um sub-elemento válido de *list*. Na especificação mostrada na Figura 4.4, seria válido incluir um outro elemento *list* entre os elementos *methodCall*. Para ilustrar o quanto isto pode ser útil, suponhamos que na classe *Account* seja inserido um novo método chamado *credit*. O método *credit* é responsável por depositar uma quantia em dinheiro na referida conta. Com a inserção deste novo método, a ordem de chamada dos métodos de *Account* deveria obedecer a seguinte ordem de chamada:

1. *auth*
2. *debit* ou *credit*
3. *deductFee*

A partir desta nova interface da classe *Account*, sua especificação agora fica conforme mostrado na Figura 4.5. Observa-se nesta nova especificação a inserção de

um segundo elemento *list*, cujo tipo é *choice*. Isto significa que somente um dos dois métodos sob *choice* poderão ser invocados depois da invocação de *auth*. Isto é equivalente ao operador barra vertical "|" das expressões regulares. A máquina de estados equivalente a esta especificação é mostrada na Figura 4.6.

```
<class name="Account">
    <methodsCallOrder>
        <list type="sequence">
            <methodCall signature="public int auth()"/>
            <list type="choice">
                <methodCall signature="public void debit()"/>
                <methodCall signature="public void credit()"/>
            </list>
            <methodCall signature="public void deductFee()"/>
        </list>
    </methodsCallOrder>
</class>
```

Figura 4.5: Nova especificação da classe *Account*

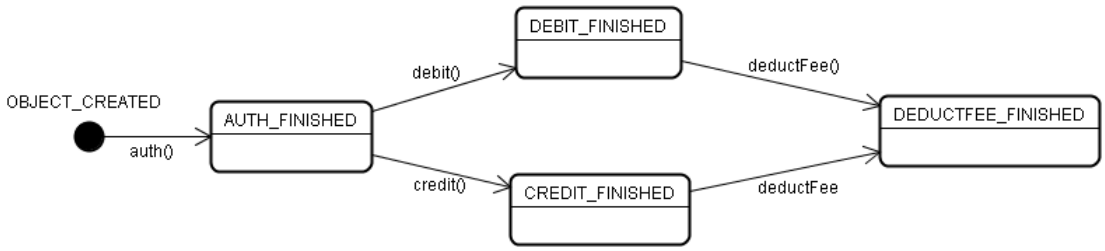


Figura 4.6: Diagrama de transição de estados para nova especificação da classe *Account*

4.3.2 Conversão de autômatos finitos não-determinísticos gerados a partir de expressões regulares para autômatos finitos determinísticos

Segundo AHO et al. (1986), a linguagem regular denotada por uma expressão regular é reconhecida por um AFN. A utilização de um AFN, entretanto, não é apropriada para controlar dinamicamente (isto é, em tempo de execução) o controle de invocação de métodos de uma classe. Prova-se que os AFNs não são apropriados para controlar a invocação de métodos de uma classe por contradição:

a) *hipótese:* para toda classe existe um AFN que controla a ordem de invocação de seus métodos corretamente;
b) se for possível provar que existe pelo menos uma classe que não é controlada corretamente por um AFN, então por contradição pode-se

inferir que a hipótese do item (a) é falsa, isto é, AFNs não são apropriados para controlar a invocação de métodos de classes.

Para provar que existe pelo menos uma classe que não pode ter os seus métodos controlados com um AFN, re-escreveu-se a especificação da classe *Account* apresentada na Figura 4.6 de maneira diferente, conforme apresentado na Figura 4.7 e na Figura 4.8, mantendo a mesma ordem de chamada de métodos (isto é, a mesma "linguagem").

```
<class name="Account">
    <methodsCallOrder>
        <list type="choice">
            <list type="sequence">
                <methodCall signature="public int auth()"/>
                <methodCall signature="public void debit()"/>
                <methodCall signature="public void deductFee()"/>
            </list>
            <list type="sequence">
                <methodCall signature="public int auth()"/>
                <methodCall signature="public void credit()"/>
                <methodCall signature="public void deductFee()"/>
            </list>
        </list>
    </methodsCallOrder>
</class>
```

Figura 4.7: Especificação da classe *Account* que gera um AFN

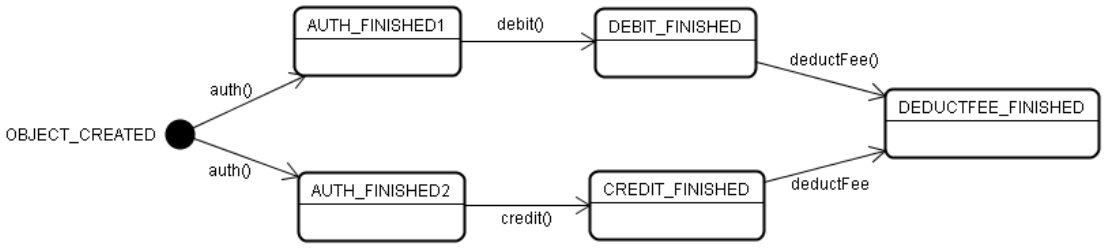


Figura 4.8: Diagrama de transição de estados para a classe *Account* que gera um AFN

O não-determinismo da máquina de estados da Figura 4.8 pode ser tratado na teoria dos autômatos finitos aplicada aos compiladores, visto que um compilador transiciona uma máquina de estados para o reconhecimento de *tokens* à medida que os símbolos da sentença que está sendo reconhecida são lidos no *buffer* de entrada (é fácil para o compilador avançar ou retroceder a posição que está sendo lida no *buffer* de

entrada). Por meio de uma técnica chamada *backtracking*[1], esta máquina de estados utilizada pelo compilador poderia ser usada para reconhecer uma linguagem regular, mesmo existindo uma indeterminação como a que há no estado OBJECT_CREATED da Figura 4.8. Neste estado, existe indeterminação pois a invocação de *auth* pode conduzir a máquina para qualquer um dos estados AUTH_FINISHED1 ou AUTH_FINISHED2.

No caso da verificação dinâmica da ordem de invocação de métodos de uma classe, o *backtracking* não é apropriado, uma vez que os métodos são invocados seguindo o fluxo normal do sistema construído sob o *framework*. Definitivamente, não poderiam haver chamadas de método adicionais para reconhecer se qualquer um dos caminhos a partir da situação de indeterminismo estariam em conformidade com a especificação de chamada de métodos da classe, assim como a técnica de *backtracking* funciona. Em outras palavras, um compilador pode facilmente avançar ou retroceder a posição de leitura do *buffer* de entrada, mas a verificação de chamada de métodos em um *framework* não pode re-executar um método ou anular o efeito de um outro método para fazer o *backtracking*.

A solução para a situação de não-determinismo consiste na conversão do autômato finito não-determinístico para um AFD (Autômato Finito Determinístico). Um AFD é um caso particular de AFN onde cada estado do AFD não possui mais de um arco de saída com o mesmo evento. AHO et al. (1986), define um AFD da seguinte forma:

> *"Um AFD é um caso especial de AFN no qual:*
> *1. nenhum estado possui uma transição* ε*, a cadeia vazia, isto é, uma transição à entrada* ε*, e*
> *2. para cada estado* s *e símbolo de entrada* a *existe no máximo um lado rotulado* a *deixando* s*." (o grifo é nosso)*

AHO et al. (1986) prova que para todo AFN que reconhece uma determinada

---

[1] O *backtracking* consiste em escolher, a partir da situação de indeterminação, um dos possíveis "caminhos" para o reconhecimento de uma sentença. Se a sentença não for reconhecida por este caminho, a máquina retorna ao estado onde a indeterminação foi encontrada e tenta o caminho seguinte, até que não existam mais caminhos. Se qualquer um dos caminhos conduzir ao reconhecimento da sentença, o AFN termina e conclui que a sentença pertence à linguagem. Caso nenhum dos caminhos reconheça a sentença, o AFN termina e conclui que a sentença não pertence à linguagem.

linguagem regular *L*, existe um AFD equivalente que também reconhece *L*. Um AFD, portanto, pode ser usado para controlar a ordem de invocação de métodos de uma classe, visto que o uso de *backtracking* não é necessário. Outra vantagem do uso dos AFDs é que estes são mais eficientes em termos de desempenho que os AFNs (AHO et al., 1995). AHO et al. (1986) também apresenta um algoritmo que executa a conversão de um AFN para um AFD equivalente. A aplicação deste algoritmo de conversão na máquina de estados não-determinística da Figura 4.8 produziria a máquina de estados determinística da Figura 4.6.

4.4 Especificação e controle do estado do sistema

Um método específico de uma classe pode ter restrições no sentido de depender da prévia execução de outros métodos, inclusive de métodos de outras classes. Define-se, para um método qualquer, em qual estado o *sistema* deve estar para que este possa ser invocado. A especificação do estado em que o sistema deve estar *antes* da invocação de um método específico chama-se *pré-condição*, ao passo que a *pós-condição* refere-se ao estado no qual o *sistema* estará após o término da execução desse método. Desta forma, a pré-condição de um método fica atrelada à pós-condição do método executado anteriormente. Esta "amarração" entre métodos de classes diferentes gera, como conseqüência, o controle do estado em que o sistema se encontra em determinado momento. A seção seguinte apresenta uma situação na qual o estado do sistema precisa ser controlado.

4.4.1 A necessidade de controlar o estado do sistema

Para ilustrar a necessidade de controlar, além do estado das classes, o estado do *sistema*, consideremos o *framework*-exemplo da Figura 4.9, que é uma versão aperfeiçoada da Figura 4.2[1].

[1] Não pretende-se apresentar aqui um *framework* completamente funcional e que atenda a todos os requisitos de uma aplicação bancária real. O objetivo é apenas ilustrar possíveis situações aonde a proposta apresentada neste trabalho se aplica.

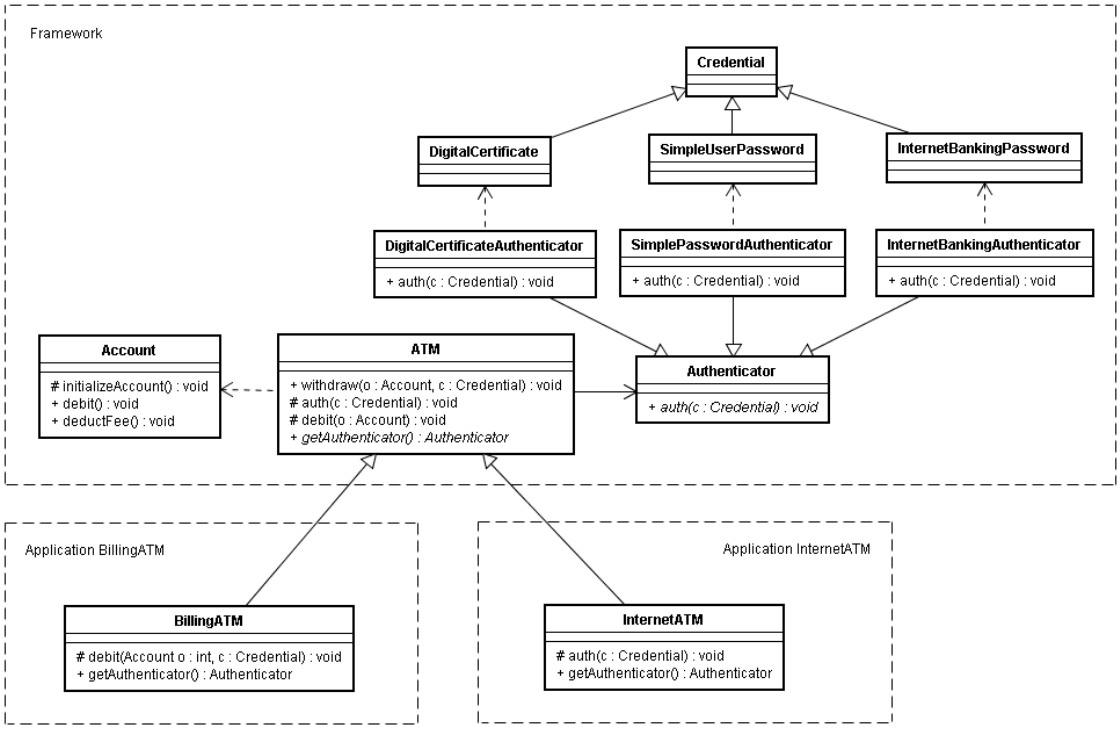


Figura 4.9: Uma versão melhorada do *framework* de sistemas bancários

No *framework* da Figura 4.9 foram inseridas as classes *Authenticator*, *Credential* e suas sub classes. Estas novas classes foram adicionadas para que o *framework* suporte uma nova funcionalidade: a autenticação do usuário de uma conta a partir de diferentes credenciais. As credenciais utilizadas por um usuário para autenticar-se podem ser: senha simples, senha estendida para uso em aplicações de Internet *banking* e certificado digital. Visto que o processo de autenticação trata-se de uma funcionalidade relativamente complexa, optou-se por remover o método *auth* da classe *Account* (conforme originalmente proposto na Figura 4.2) e delegar esta responsabilidade para a classe *Authenticator*. A classe *ATM* também foi alterada para conter um novo método *hook* chamado *auth*. Na Figura 4.10[1], encontram-se os principais métodos da classe *ATM*.

Esta nova versão do *framework* de sistemas bancários apresenta ainda uma

[1] O código da Figura 4.10 é escrito em liguagem Java, porém não é compilável. Trata-se apenas de um esboço de como alguns métodos desta classe poderiam ser implementados. O uso de reticências (...) indica o lugar no corpo do método aonde uma aplicação real eventualmente necessitaria incluir outras sentenças de código.

nova restrição ao seu uso: para que um objeto da classe *Account* possa realizar a operação de débito em uma conta, é necessário que antes o sistema *ATM* tenha autenticado o usuário que está tentando realizar a operação. Em outras palavras, a aplicação desenvolvida sobre o *framework* deverá garantir que o método *auth* de *Authenticator* seja invocado antes de *debit* de *Account*. Dito ainda de outra forma, a execução bem-sucedida do método *auth* de *Account* transiciona o estado do sistema para AUTHENTICATED, o qual é *pré-condição* do método *debit* de *Account*. Além disso, *initializeAccount* de *Account* (um método escrito para executar alguma ação de inicialização de uma conta) deve ser invocado também antes de *debit*.

```
public void withdraw(Account o, Credential) {
  …
  this.auth(…);
  o.initializeAccount();
  this.debit(…);
  o.deductFee(…);
  …
}

protected void auth(Credential) {
  Authenticator authenticator = getAuthenticator();
  authenticator.auth(c);
}

protected void debit(Account o) {
  …
  o.debit();
  …
}

public abstract Authenticator getAuthenticator();
```

Figura 4.10: Código fictício de alguns métodos da classe *ATM*

As duas aplicações desenvolvidas sobre o *framework* ilustradas na Figura 4.9, *BillingATM* e *InternetATM,* apresentam um risco em potencial: as duas sobrescrevem métodos que podem quebrar as restrições ao uso da classe *Account*. *BillingATM* deverá sobrescrever *debit* de forma tal que *debit* de *Account* seja invocado uma vez somente. Por outro lado, *auth* de *InternetATM* deverá também chamar *auth* de *Authenticator* uma vez, e garantir que sua execução seja bem sucedida, caso contrário nenhuma importância deverá ser debitada de nenhuma conta bancária. Se o usuário do *framework* não observar estas restrições, o *framework* não estará sendo bem usado e poderá apresentar um comportamento inesperado.

4.4.2 Especificação formal dos estados do sistema com arquivos XML

A análise das restrições ao uso da nova versão do *framework* apresentada na seção anterior conduz aos seguintes princípios:

a) cada método de uma classe pode ter associado a ele uma *pré-condição*, a qual é especificada em função do estado do sistema; isto significa que, se *s* é pré-condição do método *m,* então o sistema deverá estar no estado *s* antes da chamada a *m*;
b) cada método de uma classe pode ter associado a ele uma *pós-condição*, a qual indica o estado em que o sistema estará *depois* do término da execução deste método;
c) os estados de um sistema são enumerados através de um conjunto finito, onde cada elemento indica o estado do sistema que sinaliza quais métodos restritos a pré-condições podem ser invocados;
d) o estado inicial do sistema é um elemento que pertence ao conjunto definido no item (c) e que indica em qual estado o sistema está no momento em que este inicia sua execução.

O esquema XML listado no Anexo 1 dá suporte à especificação: dos estados do sistema, do estado inicial do sistema e da pré e pós-condição de cada método referenciado na especificação de classes. Ainda seguindo o esquema do Anexo 1, a Figura 4.11 mostra a nova especificação XML do *framework* de sistemas bancários.

Na especificação da Figura 4.11 listam-se os estados do sistema sob o elemento *systemStates*. Um dos sub-elementos de *systemStates* deve ser sinalizado com a propriedade *initial="yes"*, indicando que este é o estado inicial do sistema. Sob os elementos *class* também foi adicionado um elemento *methodsRestrictions*, o qual especifica para cada método quais são as suas pré e pós-condições.

```
<?xml version="1.0" encoding="UTF-8"?>
<systemSpecification
    xmlns="http://www.motorola.com/systemspecification"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.motorola.com/systemspecification
D:\rechia\systemvalidation\schemas\system_schema.xsd"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xlink:type="extended">
```

(Figura 4.11: continua na próxima página)

(Figura 4.11: continuação da página anterior)

```
  <systemStates>
      <state xlink:label="IDLE" initial="yes"/>
      <state xlink:label="AUTHENTICATED"/>
      <state xlink:label="OPERATION_COMPLETED"/>
  </systemStates>
    <class name="Account">
      <methodsRestrictions>
          <method signature="public void debit()">
              <preConditionState>
                  <state xlink:to="AUTHENTICATED"/>
              </preConditionState>
              <postConditionState>
                  <state xlink:to="OPERATION_COMPLETED"/>
              </postConditionState>
          </method>
          <method signature="public void deductFee()">
              <preConditionState>
                  <state xlink:to="OPERATION_COMPLETED"/>
              </preConditionState>
              <postConditionState>
                  <state xlink:to="IDLE"/>
              </postConditionState>
          </method>
      </methodsRestrictions>
        <methodsCallOrder>
            <list type="sequence">
                <methodCall signature="public void initializeAccount()"/>
                <methodCall signature="public void debit()"/>
                <methodCall signature="public void deductFee()"/>
            </list>
        </methodsCallOrder>
    </class>
    <class name="Authenticator">
      <methodsRestrictions>
          <method signature="public void auth(Credential)">
              <preConditionState>
                  <state xlink:to="IDLE"/>
              </preConditionState>
              <postConditionState>
                  <state xlink:to="AUTHENTICATED"/>
              </postConditionState>
          </method>
      </methodsRestrictions>
        <methodsCallOrder>
            <list type="sequence">
                <methodCall signature="public void auth(Credential)"
quantifier=”+”/>
            </list>
        </methodsCallOrder>
    </class>
</systemSpecification>
```

Figura 4.11: Especificação formal XML da nova versão do *framework* para sistemas bancários

Na versão aperfeiçoada do sistema bancário, o estado inicial do sistema é IDLE. Este estado representa a situação em que o sistema é ligado e está pronto aguardando o início da interação com o usuário. No caso do sistema *BillingATM*, o método *auth* possivelmente pede ao usuário que passe seu cartão magnético no leitor e informe o número de sua conta e senha (o comportamento de *auth* depende de como *BillingATM* implementa *getAuthenticator*; neste caso, *getAuthenticator* retorna uma instância de *SimplePasswordAuthenticator*, o qual apresenta este comportamento). Depois de terminada esta operação (isto é, após o término da execução de *auth*), o sistema passará para o estado AUTHENTICATED. Uma vez neste estado, o método *debit* de *Account* poderá ser chamado, pois a partir deste momento a pré-condição de *debit* é satisfeita. De forma análoga, uma vez executado *debit*, o sistema passará para o estado OPERATION_COMPLETED, o qual habilitará *deductFee* para ser invocado. Depois de *deductFee*, o sistema volta para IDLE e uma nova operação poderá ser iniciada.

4.4.3 O uso das redes de Petri para controle do estado do sistema

Conforme explanado nas seções 4.4.1 e 4.4.2, a ordem de invocação dos métodos de uma classe pode ser controlada com uma máquina de estados e especificada com uma expressão regular. Seguindo esta linha de raciocínio, cada classe de um *framework* pode ter associada a ela uma máquina de estados, e isso seria suficiente para controlar a ordem de chamada dos métodos desta classe. A seção anterior, no entanto, apresentou um *framework*-exemplo onde a invocação de um método específico de uma classe depende também da execução prévia do método de uma outra classe. Nesse caso, o uso de uma máquina de estados independente para cada classe não é suficiente para modelar a dependência entre métodos de classes diferentes.

Como dito no capítulo 2, as redes de Petri foram originalmente projetadas para modelar a comunicação entre autômatos finitos. Em SILVA (2002), as redes de Petri que modelam componentes de *software* genéricos são unificadas para verificar a compatibilidade da composição desses componentes. Baseado no propósito original pelo qual as redes de Petri foram criadas, a saber, a comunicação entre autômatos, e a abordagem de SILVA (2002) para a validação da composição de componentes, utilizou-se neste trabalho as redes de Petri para fazer a modelagem da dependência entre

métodos de classes diferentes aplicado aos *frameworks* orientados a objetos. Define-se, na seqüência, como usar as redes de Petri para alcançar esse objetivo.

4.4.3.1 Definindo algebricamente redes de Petri e marcação para controlar o estado de *frameworks* orientados a objetos

A partir da especificação XML das classes de um *framework* seguindo o esquema do Anexo 1, pode-se gerar uma rede de Petri conforme a definição do capítulo 2, seção 2.2.2. O disparo das transições representará a invocação de métodos dos objetos do *framework*, os lugares representarão o estado do objeto de uma classe ou o estado do sistema, e as fichas indicarão em qual estado cada objeto está e em qual estado o sistema está. Estabelece-se abaixo algumas definições que servirão de subsídios para aplicação na quádrupla $R = (P, T, Pre, Post)$:

1. seja f um *framework*;
2. seja $C = \{c \mid c$ é classe do *framework* f que possui restrições na ordem de invocação de seus métodos$\}$;
3. seja $M = \{m \mid m$ é método de uma classe $c \in C\}$;
4. seja $D : C \rightarrow M$ a relação entre uma classe e os métodos que pertencem a esta classe;
5. seja $E = \{e \mid \forall\ c\ (c \in C)\ \exists\ e$ onde e é uma expressão regular que determina a ordem de invocação dos métodos da classe $c\}$;
6. seja $Q = \{q_1, q_2, \ldots, q_k \mid q_i$ é uma máquina de estados finitos determinísticos equivalente a $e \in E$, com $1 \leq i \leq k, k \in IN^*\}$;
7. seja $G : C \rightarrow E$ onde $G(c)$ define a expressão regular que especifica a ordem de invocação de métodos da classe c;
8. seja $H : E \rightarrow Q$ onde $H[G(c)]$ define a máquina de estados finitos determinísticos que controla a ordem de invocação dos métodos da classe c e que é equivalente a $G(c)$;
9. seja $W = \{w \mid w$ é estado do sistema$\}$;
10. $w_0 \in W$ é o estado inicial do sistema;
11. seja $X : D \rightarrow W$ onde $X(c,m)$ é o estado do sistema que é pré-condição do método m da classe c;
12. seja $Y : D \rightarrow W$ onde $Y(c,m)$ é o estado do sistema que é pós-condição

do método *m* da classe *c*.

Para fins de clareza de escrita, considera-se a seguinte notação:

13. se $q \in Q$, então K(q) é o conjunto de estados de q;
14. se $q \in Q$, então $s_0(q)$ é o estado inicial de q;
15. se $q \in Q$, então $\Sigma(q)$ é o alfabeto de q;
16. se $q \in Q$, então $\Delta(q)$ é a função de mapeamento de q;
17. se A é um conjunto qualquer, então $n(A)$ refere-se à dimensão (o número de elementos) de A.

De posse das definições acima, especifica-se a rede de Petri $R = (P, T, Pre, Post)$ da seguinte forma:

a) definição do conjunto P de lugares:
   - $P = W \cup K(q_1) \cup K(q_2) \cup ... \cup K(q_k)$
     onde $k = n(Q)$, $q_i \in Q$, $1 \leq i \leq k$;

b) definição do conjunto T de transições:
   - $T = \Sigma(q_1) \cup \Sigma(q_2) \cup ... \cup \Sigma(q_k)$
     onde $k = n(Q)$, $q_i \in Q$, $1 \leq i \leq k$;

c) definição da aplicação de entrada das transições:
   - $\forall\, q\ (q \in Q), \forall\, s_1\ (s_1 \in K(q)), \forall\, s_2\ (s_2 \in K(q))$,
     $\forall\, m\ (m \in \Sigma(q)), ((s_1,m),s_2) \in \Delta(q) \rightarrow Pre(s_1,m) = 1$;
   - $\forall\, c\ (c \in C), \forall\, m\ (m \in M), \forall\, w\ (w \in W)$,
     $((c,m),w) \in X \rightarrow Pre(w,m) = 1$;
   - $Pre(p,t) = 0$ para todos os outros casos;

d) definição da aplicação de saída das transições:
   - $\forall\, q\ (q \in Q), \forall\, s_1\ (s_1 \in K(q)), \forall\, s_2\ (s_2 \in K(q))$,
     $\forall\, m\ (m \in \Sigma(q)), ((s_1,m),s_2) \in \Delta(q) \rightarrow Post(s_2,m) = 1$;
   - $\forall\, c\ (c \in C), \forall\, m\ (m \in M), \forall\, w\ (w \in W)$,
     $((c,m),w) \in Y \rightarrow Post(w,m) = 1$;
   - $Post(p,t) = 0$ para todos os outros casos.

A relação μ da marcação inicial $N = (R, \mu)$ é definida por:

- $\forall\, p\ (p \in \mathrm{P}), \forall\, q\ (q \in Q), s_0(q) = p \rightarrow \mu(p) = 1$;
- $\mu(w_0) = 1$;
- $\mu(p) = 0$, para todos os outros casos.

4.4.3.2 Interpretação da definição algébrica de redes de Petri para *frameworks* através do exemplo do sistema bancário

Os dados necessários para a criação dos conjuntos definidos nos itens numerados de 1 a 12 da seção 4.4.3.1 são extraídos de uma especificação XML que segue o esquema do Anexo 1. A fim de facilitar o entendimento da aplicação destas definições para a criação de uma rede de Petri que controla o estado de um *framework*, utiliza-se a seguir o *framework* de aplicações bancárias conforme modelado na Figura 4.9 e especificado na Figura 4.11.

Analisando a especificação XML da Figura 4.11, pode-se escrever facilmente o seguinte:

1. f é o *framework* de aplicações bancárias;
2. o conjunto de classes de f é extraído dos elementos *class* e é $C = \{Account, Authenticator\}$;
3. o conjunto de métodos é $M = \{initializeAccount, debit, deductFee, auth(Credential)\}$;
4. o conjunto que relaciona uma classe com os seus métodos é $D = \{(Account, initializeAccount), (Account, debit), (Account, deductFee), (Authenticator, auth(Credential))\}$;
5. o conjunto de expressões regulares é extraído dos elementos *methodsCallOrder*: $E = \{ initializeAccount\ debit\ deductFee, auth(Credential)^{+}\}$;
6. as máquinas de estado que pertencem ao conjunto Q estão graficamente especificadas na Figura 4.12 e Figura 4.13;

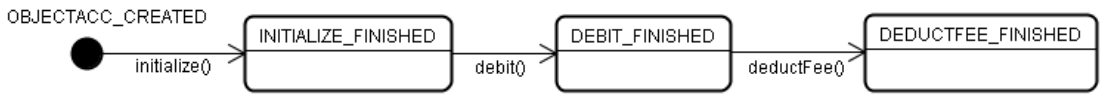


Figura 4.12: Máquina de estados da classe *Account*

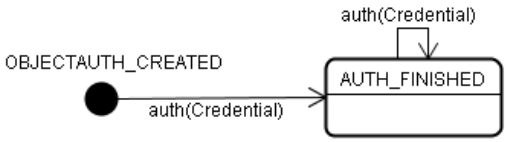


Figura 4.13: Máquina de estados da classe *Authenicator*

7. o conjunto *G* que relaciona uma classe à expressão regular que rege a chamada de seus métodos é
   G = {(*Account*, *initializeAccount debit deductFee*),
   (*Authenticator*, *auth(Credential)*$^{+}$)};
8. o conjunto *H* que relaciona uma expressão regular com a máquina de estados finitos determinísticos equivalente é
   G = {(*initializeAccount debit deductFee*, Figura 4.12),
   (*auth(Credential)*$^{+}$, Figura 4.13)};
9. os estados do sistema são extraídos do elemento *systemStates* e são
   W = {IDLE, AUTHENTICATED, OPERATION_COMPLETED};
10. o estado inicial é indicado em XML pela propriedade *initial="yes"* e é w_0 = IDLE;
11. o conjunto *X* que especifica a pré-condição de cada método é
    X = {((*Account*, *debit*), AUTHENTICATED),
    ((*Account*, *deductFee*), OPERATION_COMPLETED),
    ((*Authenticator*, *auth(Credential)*), IDLE) };
12. o conjunto *Y* que especifica a pós-condição de cada método é
    Y = {((*Account*, *debit*), OPERATION_COMPLETED),
    ((*Account*, *deductFee*), IDLE),
    ((*Authenticator*, *auth(Credential)*), AUTHENTICATED) };

Para a construção da rede de Petri que controla este *framework*, procede-se da seguinte forma:

a) o conjunto de lugares *P* é dado pela união dos estados do sistema (conjunto *W*) com os estados de cada máquina de estados finitos do conjunto *Q*:
   - *P* = {IDLE, AUTHENTICATED, OPERATION_COMPLETED,

OBJECTACC_CREATED, INITIALIZE_FINISHED, DEBIT_FINISHED, DEDUCTFEE_FINISHED, OBJECTAUTH_CREATED, AUTH_FINISHED};

b) o conjunto de transições *T* é dado pela união dos alfabetos de cada máquina de estados finitos do conjunto *Q*:
   - *T* = {*initializeAccount*, *debit*, *deductFee*, *auth(Credential)*};

c) para todo arco com rótulo *m*, em uma máquina de estados do conjunto *Q*, que parte de s_1 e chega em s_2, define-se um arco na rede de Petri que parte de s_1 e chega em *m*; se *w* é pré-condição de *m*, então define-se um arco na rede de Petri que parte de *w* e chega em *m;*

d) para todo arco com rótulo *m*, em uma máquina de estados do conjunto *Q*, que parte de s_1 e chega em s_2, define-se um arco na rede de Petri que parte de *m* e chega em s_2; se *w* é pós-condição de *m*, então define-se um arco na rede de Petri que parte de *m* e chega em *w;*

A marcação inicial é definida assim:

- os lugares que correspondem ao estado inicial de cada máquina de estados do conjunto *Q* recebem uma ficha, que são {OBJECTACC_CREATED, OBJECTAUTH_CREATED};
- o lugar que corresponde ao estado inicial do sistema w_0 = IDLE recebe uma ficha;
- todos os demais lugares não recebem nenhuma ficha.

A montagem da Rede de Petri conforme aplicação da definição algébrica da seção 4.4.3.1 pode ser ilustrada conforme Figura 4.14, Figura 4.15, Figura 4.16 e Figura 4.17.

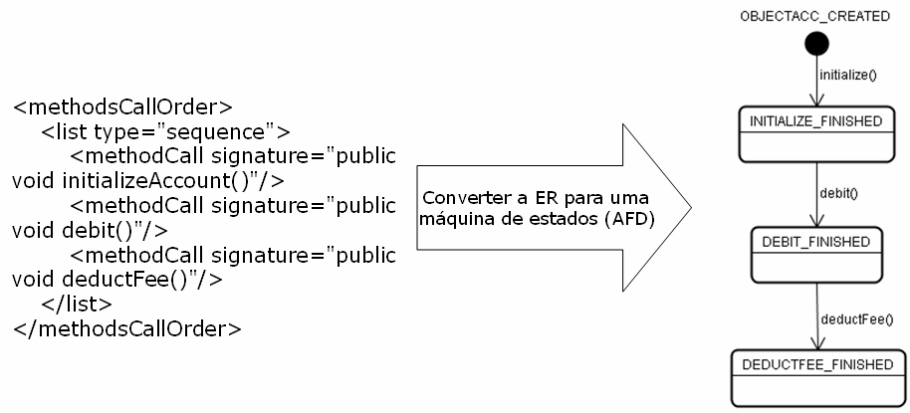


Figura 4.14: Passo 1 – Construção da Rede de Petri

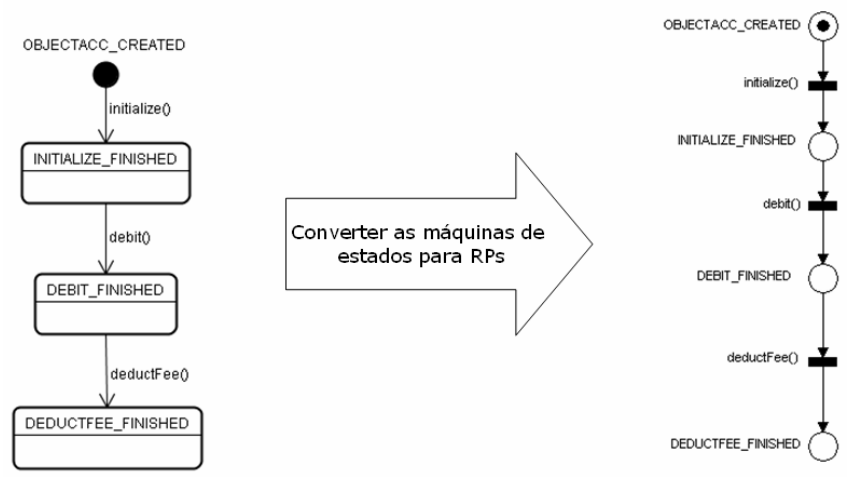


Figura 4.15: Passo 2 – Construção da Rede de Petri

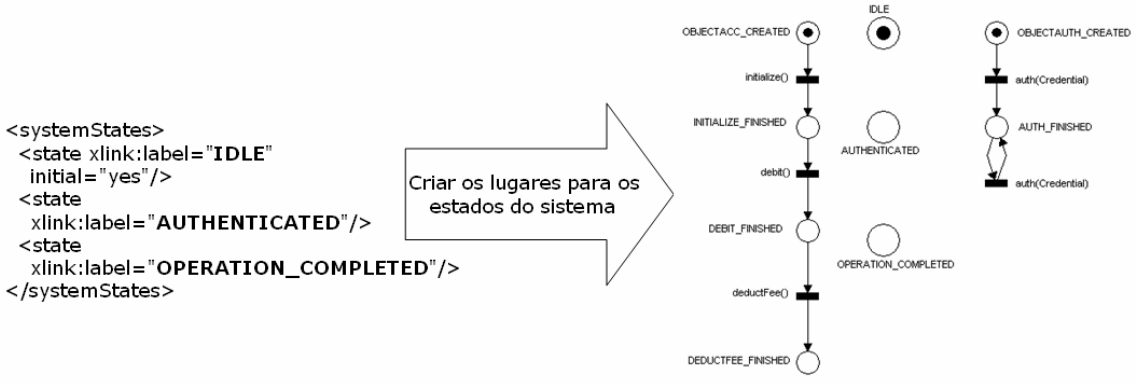


Figura 4.16: Passo 3 – Construção da Rede de Petri

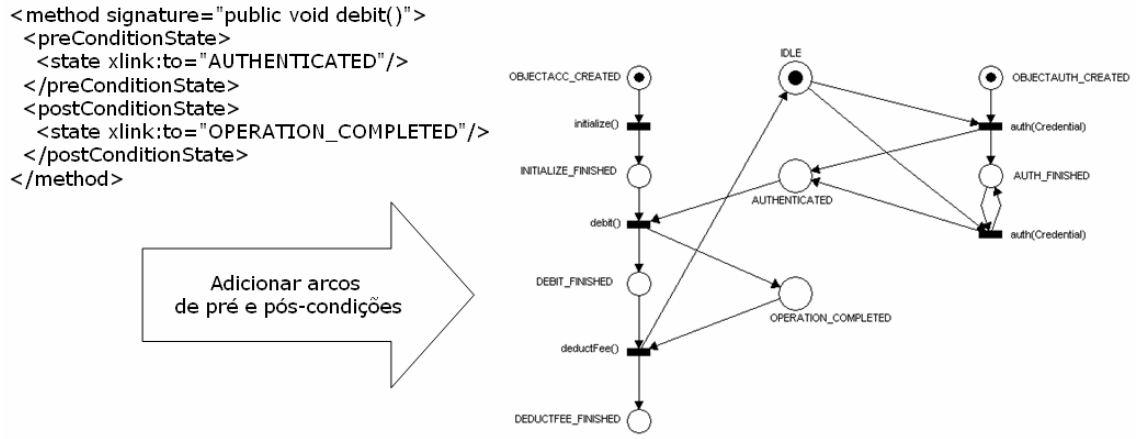


Figura 4.17: Passo 4 – Construção da Rede de Petri

Na Figura 4.14 converte-se a expressão regular em formato XML para uma máquina de estados finita; conforme considerado na seção 4.3.2, essa conversão é realizada pela execução de um algoritmo apresentado por AHO et al. (1986). A Figura 4.15 ilustra a conversão da máquina de estados para uma rede de Petri: substitui-se cada transição da máquina de estados por uma transição na rede de Petri; utiliza-se um lugar para cada estado corresponde da máquina de estados; coloca-se uma ficha no lugar que corresponde ao estado inicial. O terceiro passo, ilustrado na Figura 4.16, consiste na criação de um lugar para cada estado do sistema. Finalmente, no quarto passo, criam-se os arcos que ligam os estados do sistema às pré e pós-condições de cada método, segundo retratado na Figura 4.17.

A rede de Petri construída na Figura 4.17 está ampliada na Figura 4.18. Observa-se na Figura 4.18 que cada transição é associada com o método de uma classe. Cada lugar é associado com o estado do sistema ou com o estado de uma máquina de estados associada a uma classe do *framework*. Conforme explanado no capítulo 2, seção 2.2.2, uma transição está habilitada se o número de fichas em cada um dos lugares de entrada for maior ou igual ao peso do arco que liga este lugar à transição.

Na Figura 4.18, as fichas estão representando a marcação inicial do sistema. Na marcação inicial, as transições habilitadas correspondem aos métodos *initialize* de *Account* e *auth* de *Authenticator*; nenhum outro método pode ser invocado neste momento. Uma vez que os métodos *initialize* e *auth* são invocados (não importa qual dos dois tenha sido chamado primeiro), os lugares INITIALIZE_FINISHED, AUTH_FINISHED e AUTHENTICATED conterão uma ficha cada. Neste momento, as

única transição habilitada é a que corresponde ao método *debit* de *Account*. Depois da execução de *debit* de *Account*, os lugares DEBIT_FINISHED e OPERATION_COMPLETED conterão uma ficha cada; isto habilitará a transição que corresponde ao método *deductFee*. Finalmente, depois da execução de *deductFee*, DEDUCTFEE_FINISHED receberá uma ficha e o sistema retornará ao estado IDLE.

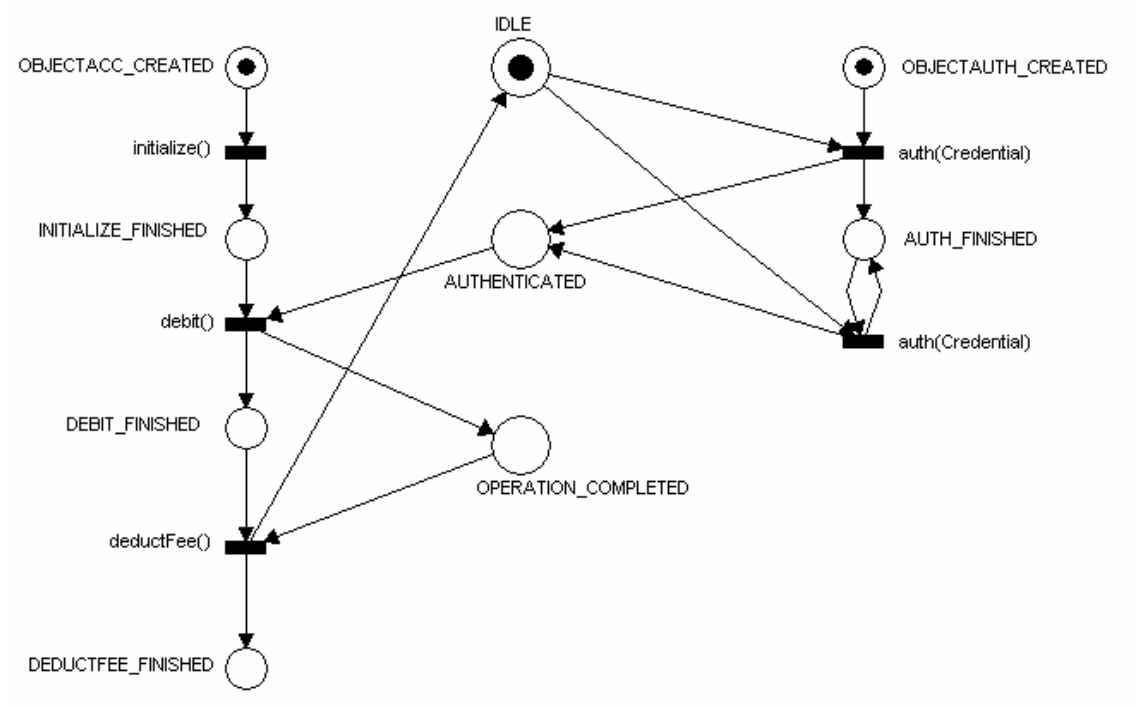


Figura 4.18: Representação gráfica da rede de Petri do sistema bancário

Vale ressaltar que a criação da rede de Petri a partir de uma especificação XML que segue o esquema do Anexo 1 pode ser criada automaticamente, de forma automatizada. Observa-se neste momento a vantagem do uso da especificação formal baseada em XML proposta neste trabalho: o desenvolvedor de um *framework* precisa se preocupar apenas em escrever a especificação XML: dos estados do sistema, das classes que possuem restrições à ordem de chamada de seus métodos, e das pré e pós-condições de cada método. Uma vez realizada esta etapa, pode-se gerar automaticamente o código que efetuará a evolução das marcações que controlarão o estado do sistema. O usuário do *framework* (aquele que o utiliza para criar uma aplicação completa) terá a garantia de que o *framework* não está sendo utilizado de forma indevida; além disso, nenhum código que verifique isso precisará ser criado manualmente. Ainda neste capítulo

descreve-se como é gerado código automático para realizar esta verificação.

4.4.4 Suporte para eventos com restrições em função de múltiplos estados

A rede de Petri da Figura 4.18 possui uma característica forte: o sistema sempre se comporta em um ciclo onde os estados do sistema sempre seguem a ordem IDLE, AUTHENTICATHED, OPERATION_COMPLETED. Uma vez em OPERATION_COMPLETED, o sistema retornará para IDLE após a execução de *deductFee* de *Account* e o ciclo se repete. Na situação real, entretanto, este ciclo nem sempre se comporta assim.

Suponhamos que, independentemente de qual estado a ATM esteja, o usuário tenha acesso a uma tecla "CANCELAR OPERAÇÃO", a qual aborta a operação iniciada pelo usuário e retorna a ATM para o estado inicial. Neste caso, a rede de Petri deveria permitir que o lugar IDLE receba uma ficha de volta, mesmo que o sistema esteja no estado AUTHENTICATED. Em outras palavras, o evento que cancela a operação atual sempre faz o sistema retornar para IDLE, não importa em qual estado o sistema esteja.

Outra limitação da rede de Petri da Figura 4.18 é o fato de que ao final da execução do método *auth* de *Authenticator* o lugar AUTHENTICATED sempre receberá uma ficha. Isto significa que o usuário *sempre* será autenticado, o que pode não ser verdade se o usuário falhar em entrar com as credenciais para acessar sua conta (como, por exemplo, na situação em que o usuário entra com a senha errada).

Para resolver as limitações expostas acima, o esquema XML do Anexo 1 suporta o uso de pré e pós condições que são especificadas em termos de múltiplos estados. Os elementos que suportam esta funcionalidade são *preConditionAnyState*, *preConditionAnyStateOf, postConditionAnyState* e *postConditionAnyStateOf*. Se a pré-condição de um método é *preConditionAnyState*, isto significa que o sistema pode estar em qualquer estado para que este método seja invocado. O elemento *preConditionAnyStateOf* permite especificar um subconjunto do conjunto dos estados do sistema; se a pré-condição de um método *m* qualquer for especificada desta forma, isto significa que o sistema deverá estar em qualquer um dos estados especificados por *preConditionAnyStateOf* para que *m* possa ser executado. De forma análoga, se a pós-condição de um método for *postConditionAnyState*, isto significa que após a execução

deste método o sistema poderá estar em qualquer estado. Finalmente, *postConditionAnyStateOf* especifica quais estados o sistema poderá estar depois da execução de um método particular.

Fazendo uso dos elementos expostos no parágrafo anterior, especifica-se agora a classe *Authenticator* e a classe *KeyboardController* (a classe que contém o método *cancelOperation* que é invocado quando a tecla "CANCELAR OPERAÇÃO" é pressionada) na Figura 4.19.

```
<class name="Authenticator">
  <methodsRestrictions>
    <method signature="public void auth(Credential)">
      <preConditionState>
        <state xlink:to="IDLE"/>
      </preConditionState>
      <postConditionAnyStateOf>
        <state xlink:to="IDLE"/>
        <state xlink:to="AUTHENTICATED"/>
      </postConditionAnyStateOf>
    </method>
  </methodsRestrictions>
  <methodsCallOrder>
    <list type="sequence">
      <methodCall signature="public void auth(Credential)"
quantifier="+"/>
    </list>
  </methodsCallOrder>
</class>

<class name="KeyboardController">
  <methodsRestrictions>
    <method signature="public void cancelOperation()">
      <preConditionAnyState/>

      <postConditionState>
        <state xlink:to="IDLE"/>
      </postConditionState>
    </method>
  </methodsRestrictions>

  <methodsCallOrder>
    <list type="sequence">
      <methodCall signature="public void cancelOperation()"
quantifier="+"/>
    </list>
  </methodsCallOrder>
</class>
```

Figura 4.19: Nova especificação de *Authenticator* e especificação de *KeyboardController*

A rede de Petri que deve ser montada conforme especificação das classes

Authenticator e *KeyboardController* da Figura 4.19 é mostrada na Figura 4.20. Observa-se nesta rede de Petri que o disparo de qualquer uma das transições *disableIdle, disableAuth* ou *disableOper* retirará a ficha do respectivo estado do sistema e colocará uma ficha em PRE_ANY_STATE. Nesta situação, o método *cancelOperation* poderá ser invocado e o sistema retorna para IDLE (isto é, a tecla "CANCELAR OPERAÇÃO" poderá ser pressionada, independentemente do estado em que o sistema está). Observa-se também que a invocação de *auth* de *Authenticator* sempre coloca uma ficha no lugar POST_ANY_OF_IDLE_AUTH. Existe neste caso um conflito: as transições *goToIdle* e *goToAuth* estão habilitadas e disputando uma ficha. Se o resultado da operação de autenticação for positivo, *goToAuth* deve ser disparada; em caso contrário, deve-se disparar *goToIdle*.

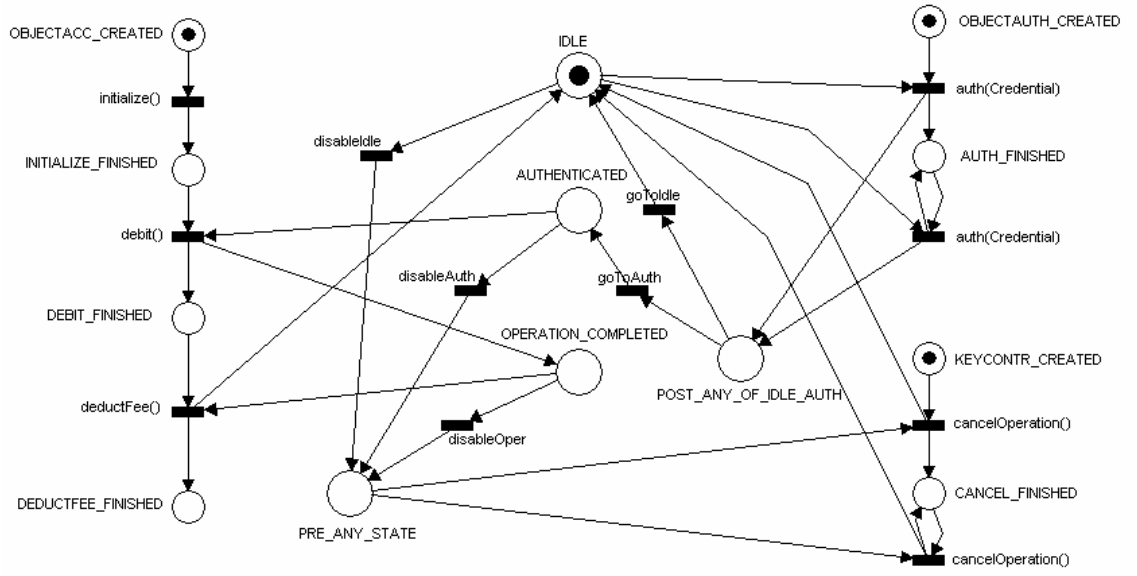


Figura 4.20: Rede de Petri para o suporte de restrições em função de múltiplos estados

O uso dos elementos *preConditionAnyState*, *preConditionAnyStateOf, postConditionAnyState* e *postConditionAnyStateOf* exigem a redefinição de alguns conjuntos e relações definidos na seção 4.4.3.1. Informalmente, descreve-se abaixo o procedimento adicional ao descrito na seção 4.4.3.1 para a criação da rede de Petri com suporte a restrições em função de múltiplos estados:

a) seja ρ(*W*) o conjunto potência de *W* (*W* é o conjunto dos estados do sistema conforme descrito na seção 4.4.3.1) definido como

$\rho(W) = \{A \mid A \subseteq W\}$;

b) seja $W_i \in \rho(W)$, onde W_i é definido no elemento *preConditionAnyStateOf* de um método *m* qualquer;

c) seja $W_o \in \rho(W)$, onde W_o é definido no elemento *postConditionAnyState* de um método *m* qualquer;

d) para todo W_i definido no item (b): criar um novo lugar *p*; criar uma nova transição para cada $w \in W_i$ e ligar estas transições com um arco partindo de cada nova transição para o novo lugar *p*; ligar cada lugar $w \in W_i$ com um arco partindo de *w* e chegando na nova transição respectiva; se a pré-condição de um método *m* qualquer for escrita com o elemento *preConditionAnyStateOf* tal que os estados indicados neste elemento formem o conjunto W_i, então criar um arco que parte de *p* e chega em *m*;

e) para todo W_o definido no item (c): criar um novo lugar *p*; criar uma nova transição para cada $w \in W_o$ e ligar estas transições com um arco partindo de *p* e chegando em cada nova transição; ligar cada nova transição com um arco partindo desta transição e chegando no respectivo estado *w*; se a pós-condição de um método *m* qualquer for escrita com o elemento *postConditionAnyStateOf* tal que os estados indicados neste elemento formem o conjunto W_o, então criar um arco na rede de Petri que parte de *m* e chega em *p*;

f) para o elemento *preConditionAnyState*, procede-se conforme descrito no item (d), onde considera-se o caso particular onde $W_i = W$;

g) para o elemento *postConditionAnyState*, procede-se conforme descrito no item (e), onde considera-se o caso particular onde $W_o = W$.

4.5 Verificação dinâmica de conformidade do sistema com a especificação formal utilizando aspectos

A utilização das redes de Petri para controlar a ordem das chamadas de métodos dos objetos de um *framework* possui um desafio: como disparar as transições

da rede de Petri à medida em que os métodos do *framework* são chamados? Se utilizada somente a abordagem orientada a objetos pura, seria necessária a inclusão de código em cada método do *framework* para que uma mensagem seja enviada à rede de Petri, indicando a ela que a transição que corresponde a este método deve ser disparada. Essa abordagem possui vários incovenientes: as classes do *framework* deverão possuir uma referência para o objeto que representa a rede de Petri, mesmo que a rede de Petri não constitua um objeto que pertence ao domínio que está sendo tratado pelo *framework*; cada método associado a uma transição da rede de Petri deverá ser alterado para disparar esta transição, isto significa que o projetista e/ou programador do *framework* deverá lembrar-se sempre de verificar isto quando novos métodos forem acrescentados no *framework*; o código que dispara a transição das redes de Petri ficará embutido no código do *framework* – isto tornará a manutenção difícil e o código ficará menos legível.

Nesse contexto, a inclusão de aspectos torna-se apropriada. O uso de aspectos para a validação de sistemas tem precedentes nos trabalhos de UBAYASHI & TETSUO (2002) e LAM et al. (2005) e serviram de base para o uso de aspectos no contexto deste trabalho. Seguindo a abordagem orientada a aspectos, o disparo das transições de uma rede de Petri pode ser considerado uma área de interesse; pontos de corte podem ser definidos antes ou depois da execução de métodos específicos de um *framework* (dependendo do caso) para disparar a transição apropriada da rede de Petri. O código que instancia a rede de Petri e o código que dispara suas transições ficam centralizados em um único módulo, não sendo necessária a inclusão de código embutido no *framework* para realizar esta tarefa. Outra vantagem do uso de aspectos neste caso é o fato de que, uma vez que o sistema está validado, pode-se facilmente compilar novamente a aplicação final sem os aspectos; isto pode ser feito se o disparo de transições da rede de Petri constituir um problema para o desempenho da execução do sistema. Vale ainda ressaltar que nem o código do *framework* nem o código que estende este *framework* para constituir a aplicação final serão modificados para que a verificação da chamada dos métodos seja efetuada.

4.5.1 Geração automática de aspectos para o disparo de transição de redes de Petri

Pode-se gerar automaticamente um aspecto que controla a chamada dos métodos dos objetos de um *framework* através de uma rede de Petri que é montada conforme indicado nas seções 4.4.3.1 e 4.4.4, tendo como entrada um arquivo XML escrito seguindo o esquema listado no Anexo 1. A criação deste aspecto é fundamentada de acordo com os seguintes princípios:

a) os estados do sistema são únicos e haverá apenas um objeto criado em memória segundo o padrão de projeto *Singleton* (GAMMA et al., 1994) para conter os objetos que representam cada estado;
b) cada vez que um objeto de uma classe que pertence ao conjunto *C* (o conjunto de classes definido na seção 4.4.3.1) ou cada vez que um objeto de qualquer subclasse do conjunto *C* for criado, será instanciada uma "sub-rede" de Petri em memória que controlará a ordem de invocação dos métodos deste objeto (esta sub-rede é criada com base na máquina de estados que pertence ao conjunto *Q*, a qual é criada a partir da expressão regular equivalente pertencente ao conjunto *E* e que é extraída da especificação XML); se qualquer um dos métodos controlados por essa sub-rede possuir restrições associadas à invocação de métodos de outras classes diferentes (isto é, se houverem métodos com pré ou pós-condições), serão criados arcos que ligarão as transições associadas a estes métodos com os respectivos lugares que representam os estados do sistema conforme especificado no item (a);
c) o aspecto gerado deverá interceptar o fluxo normal de execução do *framework* para disparar a transição associada a cada método somente se essa transição estiver habilitada; em caso contrário, este método não deve ser executado e o usuário deverá ser advertido que este método foi invocado no momento indevido;

A fim de exemplificar os princípios colocados acima, observe a Figura 4.21, a qual destaca as sub-redes e os estados do *framework* de sistemas bancários da Figura

4.20. Observa-se na Figura 4.21 que os lugares e transições contidos dentro do retângulo rotulado “Sub-rede para Account” é equivalente à máquina de estados da Figura 4.12, exceto pelos arcos que se ligam aos lugares dos estados do sistema. Da mesma forma, os lugares e transições contidos no retângulo rotulado “Sub-rede para Authenticator” são equivalentes à máquina de estados da Figura 4.13.

No *framework* de sistemas bancários, cada vez que um objeto da classe *Account* for criado, uma nova sub-rede indicada para esta classe na Figura 4.21 será criada e anexada aos lugares que correspondem aos estados do sistema. Da mesma forma, também haverá uma sub-rede para cada objeto das classes *Authenticator* e *KeyboardController*.

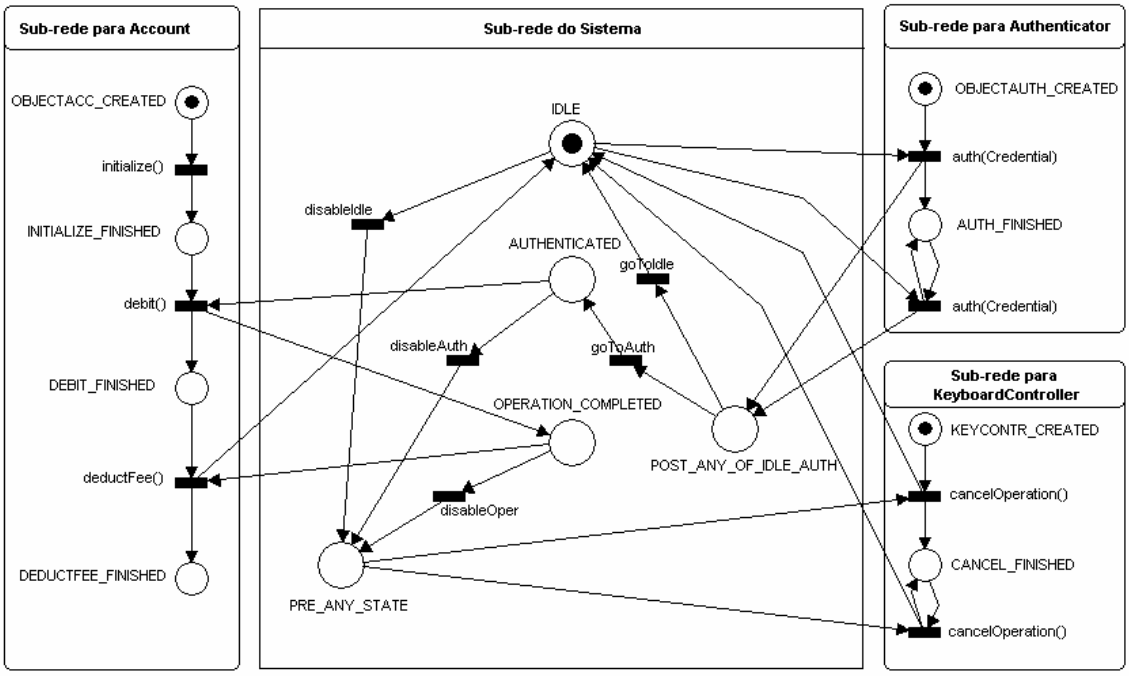


Figura 4.21: Sub-redes da rede de Petri do *framework* de sistemas bancários

Vale salientar, entretanto, que este modelo prevê apenas uma instância de cada subrede em um determinado instante, isto é, não poderão haver, ao mesmo tempo, múltiplas instâncias da sub-rede para *Account* e múltiplas instâncias da sub-rede para *Authenticator*. Por exemplo: consideremos novamente a Figura 4.21. Suponhamos que, em um sistema concorrente, existam vários objetos da classe *Account* instanciados em memória, sendo que cada objeto *Account* foi instanciado por um usuário diferente. Cada usuário, obviamente, instanciou também um objeto *Authenticator* para acessar sua conta

particular. O problema é: qual é o objeto *Authenticator* que autentica as operações para um objeto qualquer da classe *Account*?

A limitação exposta acima se dá pelo fato de que este modelo não diferencia qual instância da sub-rede de *Account* corresponde à instância da sub-rede de *Authenticator*. Em outras palavras, a especificação XML das restrições ao *framework* preocupa-se com as *classes,* e não com os *objetos* que compõem o *framework*. Como assunto de futura pesquisa, pode-se utilizar redes de Petri coloridas (CARDOSO & VALETTE, 1997), de forma tal que cada cor de ficha identificaria a correspondência entre as múltiplas instâncias que monitoram os objetos transientes.

Como resultado deste projeto de pesquisa, produziu-se uma ferramenta capaz de gerar um aspecto que lê um arquivo XML que segue o esquema do Anexo 1 e cria em tempo de execução uma rede de Petri conforme os princípios acima. Esta ferramenta produziu o aspecto em linguagem *AspectJ* listado no Anexo 2 para a especificação XML apresentada na Figura 4.11.

A ferramenta gera dois arquivos: uma classe Java e um aspecto *AspectJ*. A classe Java possui atributos estáticos (também chamados de atributos de classe) que representam os lugares que são os estados do sistema. O aspecto gerado é responsável pela montagem das sub-redes e o disparo das transições no momento em que o método é chamado.

Para o *framework* de sistemas bancários, a criação da sub-rede de *Account* é codificada no método *private void Account.initializePetriNet()* (ver Anexo 2). Neste caso, *AspectJ* acrescenta o método *initializePetriNet* na classe *Account*, o qual é responsável pela instanciação dos objetos *Place* (lugar) e *Transition* (transição). Este método também cria os arcos entre as transições e os lugares, além de colocar uma ficha no estado que indica que o objeto foi criado. A Figura 4.22 mostra o aviso de *AspectJ* (extraído do Anexo 2) que é utilizado para invocar *initializePetriNet* cada vez que um objeto desta classe é criado. Vale ressaltar que este aspecto também aplica-se a qualquer sub-classe de *Account*.

```
after(Account componentInstance):
    target(componentInstance) && initialization(Account.new())
{
    componentInstance.initializePetriNet();
}
```

Figura 4.22: Aviso para a inicialização da sub-rede de *Account*

A Figura 4.23 mostra o aviso extraído do Anexo 2 que dispara a transição apropriada da rede de Petri para o método *initializeAccount* da classe *Account*. Quando o método *initializeAccount* é chamado e a transição correspondente a ele está habilitada, o disparo é efetuado (pelo método *fire*); se a transição não estiver habilitada, uma exceção é lançada e o programa é interrompido. A exceção lançada é a *RuntimeException*, a qual deverá terminar a execução do programa no ponto onde a exceção é criada; a pilha de execução de métodos será mostrada ao programador.

```
before(Account componentInstance):
    target(componentInstance) &&
    execution(public void initializeAccount())
{
    if (componentInstance.TRANSITION_9.enabled())
    {
        componentInstance.TRANSITION_9.fire();
    }
    else {
        throw new RuntimeException(new
            com.motorola.compsvalidation.petrinet.
            PetriNetTransitionNotAllowed());
    }
}
```

Figura 4.23: Aspecto para o disparo da transição que corresponde ao método *initializeAccount* de *Account*

Conforme explanado na seção 4.4.4, existe um conflito efetivo na rede de Petri na situação onde uma ficha é colocada no lugar POST_ANY_OF_IDLE_AUTH da Figura 4.20[1]. No sistema bancário, este conflito existe e é resolvido baseado no resultado do método *auth* de *Authenticator*. Se a autenticação for bem-sucedida, basta disparar *goToAuth*; caso contrário, dispara-se *goToIdle*. No aspecto automaticamente

[1] O nome do lugar POST_ANY_OF_IDLE_AUTH no código automaticamente gerado do Anexo 2 é POST_CONDITION_ANY_STATE_OF_1.

gerado pela ferramenta desenvolvida neste trabalho, resolve-se o conflito delegando esta decisão para um objeto do tipo *AbstractIndeterminationHandler*. Observa-se no código automaticamente gerado do Anexo 2 que o aspecto *ATMValidationAspect* cria o objeto *indetermHandler* da classe *IndeterminationHandler*, a qual é sub-classe de *AbstractIndeterminationHandler*. No aviso que dispara a transição apropriada para o método *auth* de *Authenticator*, chama-se o método *indetermHandler.determineAndFireTransition*[1], o qual decidirá qual das duas transições *goToAuth*[2] ou *goToIdle*[3] deverá ser disparada. No caso do *framework* para sistemas bancários, o método *determineAndFireTransition* de *IndeterminationHandler* deverá consultar o resultado do método *auth* de *Authenticator* e disparar a transição apropriada.

Quando compilado junto com qualquer aplicação desenvolvida sobre o *framework* de sistemas bancários, o aspecto do Anexo 2 será capaz de advertir o usuário sobre alguma chamada de método que não respeite a ordem de invocação conforme especificado no arquivo XML. Vale ressaltar que esta verificação é feita somente em tempo de execução, nenhuma chamada é verificada em tempo de compilação. Isto facilitará a depuração do sistema que está sendo desenvolvido sobre um *framework* na fase em que o sistema estiver sendo testado ou desenvolvido. Embora esta abordagem não teste a ordem de invocação por si só, os testes funcionais do sistema final como um todo poderão ser auxiliados pelo método proposto. Para que a verificação de chamada de métodos seja realmente efetiva, serão necessários casos de teste que cubram amplamente o código do *software* sob avaliação.

O exemplo do *framework* de sistemas bancários apresentado até agora não constitui o código de uma aplicação real, embora não esteja muito longe de representar (ao menos uma fração de) um sistema completo. No próximo capítulo, apresentar-se-á a aplicação do método proposto em um *framework* utilizado no meio corporativo. O uso deste modelo em um software utilizado no mundo real atesta sua eficácia.

---

[1] O método *determineAndFireTransition* é abstrato na classe *AbstractIndeterminationHandler* e é concretamente implementado em *IndeterminationHandler*.

[2] O nome da transição *goToAuth* no código automaticamente gerado do Anexo 2 é TRANSITION_8.

[3] O nome da transição *goToIdle* no código automaticamente gerado do Anexo 2 é TRANSITION_7.

5 ESTUDO DE CASO: ESPECIFICANDO FORMALMENTE RESTRIÇÕES PARA O *FRAMEWORK TAF – TEST AUTOMATION FRAMEWORK*

5.1 TAF: motivação e visão geral da arquitetura

O TAF é um *framework* projetado para suportar a automação de testes funcionais dos *softwares* embutidos em telefones celulares produzidos e desenvolvidos pela Motorola Industrial Ltda. A principal motivação para o uso do TAF é a reutilização de código a partir da portabilidade de testes. Os testes que são executados em telefones de modelos diferentes, mas que implementam as mesmas funcionalidades, são os mesmos. Embora existam telefones que implementam as mesmas funcionalidades, a maneira como o usuário interage com estas funcionalidades varia em função do modelo do aparelho. Observando as funcionalidades comuns entre telefones diferentes e a elevada quantidade de testes em comum que é executada nestes aparelhos, projetou-se o TAF de forma tal que os mesmos casos de teste automatizados pudessem ser reutilizados em telefones diferentes. Além da reutilização dos casos de teste, a automação destes também reduz o tempo gasto no ciclo de desenvolvimento do *software* embutido nos telefones, eliminando a necessidade da execução manual.

Para interagir com o telefone, o TAF utiliza uma biblioteca proprietária de funções chamada PTF (*Phone Test Framework*). O PTF comunica-se com o telefone via interface USB (*Universal Serial Bus*) para prover funções que simulam o pressionamento de teclas e retornar o texto que está sendo mostrado na tela do telefone. Com o PTF é possível estimular o teclado do telefone e verificar se o conteúdo mostrado na tela do aparelho é o esperado. As funções providas pelo PTF, entretanto, são de baixo nível de abstração; a utilização direta do PTF para automatizar um caso de teste produz *scripts* de teste ilegíveis e difíceis de serem portados para serem executados em outro modelo de telefone.

O TAF foi projetado para resolver o problema do baixo nível de abstração das funções providas pelo PTF e promover a reutilização de código de testes automatizados. No TAF, a seqüência de teclas a serem pressionadas ou a verificação do conteúdo

mostrado na tela do telefone (isto é, a chamada direta a métodos do PTF) são encapsulados em UFs (*Utility Functions*). Uma UF é a implementação de um passo de alto nível que é executado em um caso de teste. Com o TAF, os casos de teste são escritos em termos de UFs de alto nível, ao invés de chamadas a funções de baixo nível, como as que o PTF provê. Alguns exemplos de UFs que representam um passo de alto nível em um caso de teste são: *Call* (faz uma chamada para o número especificado), *LaunchApp* (abre uma aplicação, como o editor de mensagens ou a lista de contatos), *CapturePictureFromCamera* (fotografa utilizando a câmera do telefone), *DeleteFile* (remove um arquivo qualquer do telefone, como uma figura, som ou vídeo).

Como mostra a Figura 5.1, as UFs são classes que implementam uma interface chamada *Step*. Sob certo ponto de vista, um caso de teste é uma lista de *Steps*; tudo o que um caso de teste faz é invocar o método *execute* que é definido na interface *Step* e implementado nas UFs concretas. Cada UF, entretanto, além de implementar a interface *Step*, possui uma API (*Application Programming Interface*) bem definida que provê métodos que possibilitam a interação com esta UF. Por exemplo, a UF *LaunchApp* possui o método *setApplication*, o qual indica à *LaunchApp* qual aplicação deverá ser iniciada quando esta UF for executada. O diagrama de classes da Figura 5.1 apresenta uma visão simplificada do TAF. Detalhes do *framework* irrelevantes à aplicação da abordagem proposta neste trabalho estão sendo suprimidos.

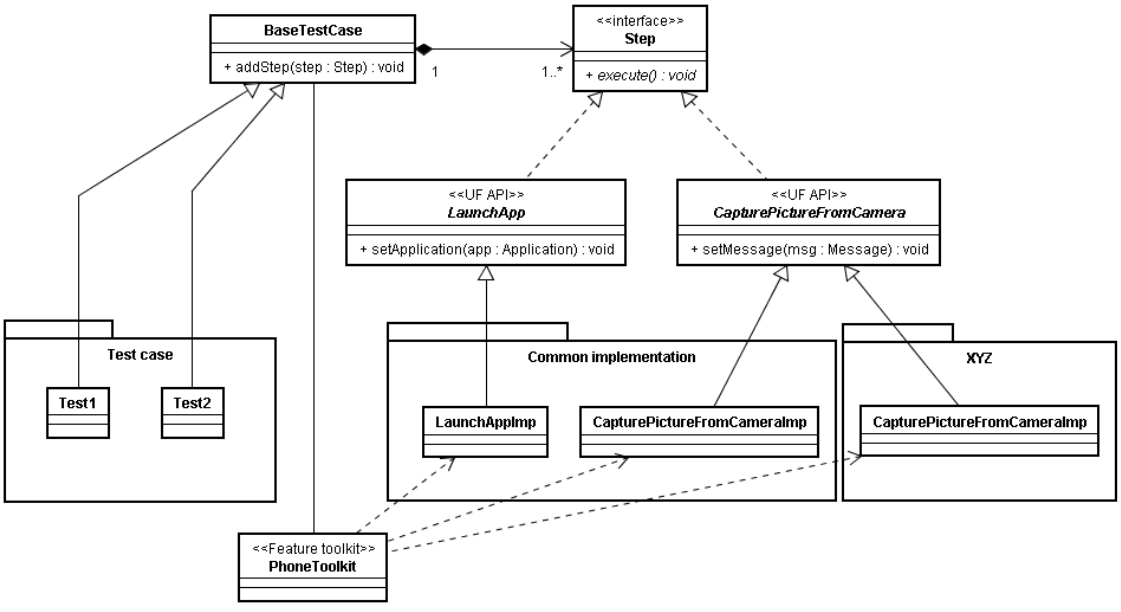


Figura 5.1: Diagrama de classes do TAF

O TAF, na qualidade de um *framework* usado no meio corporativo, é complexo e extenso: existem atualmente cerca de 420 UFs (APIs diferentes), com mais de 590 implementações destas UFS, além de rotinas adicionais cujas classes não estão representadas na Figura 5.1. Tais rotinas adicionais fornecem funcionalidades tais como: entrada de texto nos diferentes editores do telefone celular, comparação de *strings* para suportar o truncamento de textos longos mostrados na tela do telefone, gerenciamento da conexão entre o computador e o telefone, etc.

A reutilização de código dos casos de teste se dá quando um caso de teste que já é executado e funciona em um telefone é portado para outro aparelho. Se a implementação de uma UF específica não funciona no telefone para o qual o teste está sendo portado, cria-se uma nova implementação desta UF para este telefone. Por exemplo, na Figura 5.1 criou-se uma implementação específica da UF *CapturePictureFromCamera* para um telefone fictício XYZ, visto que o comportamento deste telefone particular para tirar uma fotografia é diferente dos demais. Assim, toda UF é um potencial *hot spot.* Na prática, a API de uma UF é uma classe abstrata que implementa a interface *Step*. As implementações de uma UF são classes concretas que extendem a API desta UF. Desta forma, consegue-se reutilizar um caso de teste reimplementando somente as UFs que ainda não funcionam no telefone para o qual o caso de teste está sendo portado.

Um caso de teste é composto por várias chamadas a métodos dos *feature toolkits*. Os métodos dos *feature toolkits* adicionam um *step* na lista de *steps* de um caso de teste. A responsabilidade de um *feature toolkit* resume-se em criar a instância de uma UF específica, passar a ela os argumentos necessários (por exemplo, chamar o método *setApplication* de *LaunchApp* para que esta UF seja informada de qual aplicação deve ser iniciada) e acrescentar a instância dessa UF na lista de *Steps* do caso de teste. Uma vez definida a lista de *Steps* através dos *feature toolkits*, o teste poderá ser executado.

A Figura 5.2 mostra a arquitetura em camadas utilizada pelo TAF, onde se visualiza a relação entre a camada onde estão os casos de teste, a camada onde os *feature toolkits* são implementados, a camada das UFs (API + implementação), e o PTF.

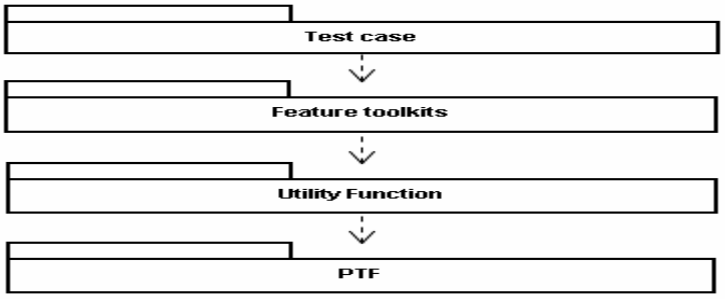


Figura 5.2: Arquitetura de camadas do TAF

5.2 Controle de execução de métodos das UFs

Existem UFs que possuem restrições quanto à ordem de chamada de seus métodos. Um exemplo simples, mas concreto, é a UF *LaunchApp*. Esta UF não pode ser executada (isto é, seu método *execute* não pode ser invocado) a menos que o método *setApplication* tenha sido invocado antes. Isto significa que para que o TAF possa iniciar uma applicação do telefone, é preciso que antes a UF *LaunchApp* saiba para qual aplicação ela tem que navegar.

As restrições à ordem de chamada dos métodos da API das UFs do TAF podem ser controladas utilizando o esquema XML do Anexo 1. Baseado neste esquema, especifica-se quais métodos precisam obrigatoriamente ser chamados antes da execução do método *execute* (o método abstrato definido na interface *Step*). O suporte para a escrita da ordem dos métodos, de forma equivalente à uma expressão regular, permite também a especificação de chamadas de métodos opcionais, cujas execuções prévias não são obrigatórias para *execute*. Ainda neste capítulo, serão fornecidos exemplos da aplicação da abordagem explanada no capítulo anterior para a ordem de chamada dos métodos das UFs.

5.3 Controle do estado do telefone

Em termos simples, a execução de uma UF produz uma mudança de estado no telefone. Cada UF possui uma pré-condição que é especificada em termos do estado em que o telefone deve estar para que sua execução seja possível. Após o término da execução de uma UF, o telefone estará no estado especificado pela pós-condição desta UF. A Figura 5.3 representa de forma esquemática o efeito que a execução de uma UF

produz no telefone.

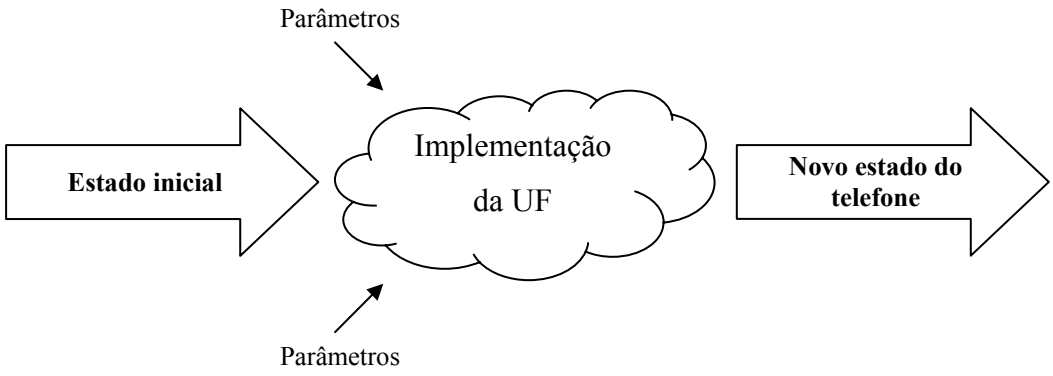
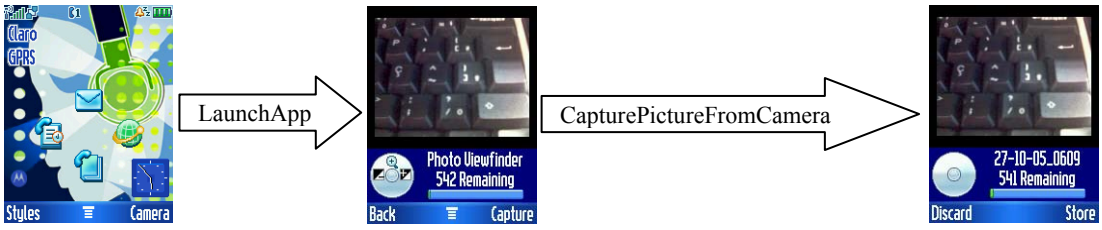


Figura 5.3: Efeito de uma UF no estado do telefone

As pré e pós-condições das UFs na maioria das vezes são especificadas em termos da tela em que o telefone deve encontrar-se. Por exemplo, uma pré-condição para a execução da UF *CapturePictureFromCamera* é que o telefone esteja na tela *Photo Viewfinder*. Depois de tirada a fotografia (isto é, depois da execução de *CapturePictureFromCamera*), o telefone perguntará ao usuário o que ele deseja fazer com a fotografia (ele poderá salvar ou descartar a fotografia, enviá-la para alguém através de uma mensagem multimídia, etc). Uma pós-condição de *CapturePictureFromCamera*, portanto, é o telefone estar nessa tela, a qual é chamada de *Save/discard screen*. Assim como *CapturePictureFromCamera*, a maioria das UFs exigem que o telefone esteja em uma tela específica para que sejam executadas, e não raro, o telefone estará em outra tela diferente, ao término da execução da UF. A Figura 5.4 mostra de forma esquemática o efeito da execução das UFs *LaunchApp* e *CapturePictureFromCamera* no telefone.



Idle *Photo Viewfinder* *Save/discard screen*

Figura 5.4: Efeito das UFs *LaunchApp* e *CapturePictureFromCamera*

A execução de uma UF pode alterar características diferentes do telefone. Em outras palavras, o efeito da execução de uma UF pode modificar muitas "variáveis" do celular. Até o momento, mencionou-se que um efeito da UF *CapturePictureFromCamera* é conduzir o telefone da tela *Photo Viewfinder* para a tela *Save/discard screen*. Este, obviamente, é apenas um dos efeitos. Após a fotografia, o telefone possuirá menos memória disponível, o sistema de arquivos do telefone será modificado (depois da fotografia, um arquivo temporário é usado para guardar a fotografia recentemente tirada) e, obviamente, haverá uma figura disponível para salvar, descartar, enviar por mensagem, e assim por diante. Cada efeito diferente produzido por uma UF, isto é, cada "variável" diferente é uma candidata potencial para compor o conjunto de pré e pós-condições de uma UF. Controlar e documentar todas as variáveis de um telefone para cada UF não é prático, para não dizer inviável, em vista da grande quantidade de variáveis que o telefone possui. Enumerar todos os estados possíveis em que o telefone pode estar em função da combinação das variáveis descritas acima é igualmente difícil. Na prática, isso significa que é preciso limitar o número de variáveis a serem monitoradas para que seja possível a aplicação da abordagem formal de controle de estados do sistema apresentado no capítulo anterior.

Em um caso de teste criado com o TAF, a ordem em que as UFs são chamadas em um caso de teste deve ser tal que a pré-condição de uma UF deve ser compatível com a pós-condição da UF que foi chamada anteriormente. Por exemplo, antes da chamada a *CapturePictureFromCamera*, é necessário que exista uma UF cuja pós-condição garanta que o telefone esteja na tela *Photo Viewfinder*. Dito de outra forma, a chamada de uma UF depende da UF que foi invocada anteriormente; se esta dependência não for respeitada, o caso de teste estará incorreto. Nesse caso, o teste falha, mesmo que o telefone apresente o comportamento esperado. Esta inconsistência será detectada pelo desenvolvedor somente depois que a execução do teste terminar e o resultado for *FAILED*. Depois disso, o desenvolvedor iniciará a árdua tarefa de depurar o código do teste automatizado, até descobrir que uma restrição à ordem de chamada de uma UF não foi respeitada, isto é, o erro está no *software* testador (o caso de teste construído sob o TAF) e não no *software* que está sendo testado (o *software* do telefone).

Para resolver esse problema, utiliza-se a abordagem proposta no capítulo

anterior. Modela-se os estados do telefone como uma representação das telas pelas quais o usuário pode navegar. Especifica-se a pré e pós-condição do método *execute* de cada UF em função destes estados, isto é, a restrição a um método *execute* de uma UF em particular é escrita em termos da tela em que o telefone precisa estar para que esta UF funcione. Uma vez definidas as pré e pós condições de cada UF juntamente com a ordem de chamada dos métodos de sua API através do esquema listado no Anexo 1, o aspecto gerado pela ferramenta produzida neste trabalho é capaz de validar formalmente cada caso de teste contruído sob o TAF.

5.4 Validando um caso de teste do TAF

O Anexo 3 lista um caso de teste construído sob o TAF. O propósito do caso de teste é tirar e salvar uma fotografia, verificar se esta foi realmente salva, verificar se os atributos da fotografia salva são mostrados corretamente (os atributos são o nome e extensão do arquivo, tamanho, etc) e finalmente apagar a fotografia ao final do teste. No Anexo 3, as chamadas aos métodos dos *feature toolkits* são colocados no método *buildProcedures* (estas chamadas estão destacadas em negrito). Conforme já explanado, cada método dos *feature toolkits* instancia a UF apropriada e adiciona esta instância na lista de *Steps* do caso de teste. Os *feature toolkits* utilizados no caso de teste do Anexo 3 são: *navigationTk* (*toolkit* de navegação), *multimediaTk* (*toolkit* de UFs de multimídia) e *phoneTk* (*toolkit* de funcionalidades inerentes ao telefone).

No caso de teste do Anexo 3 são utilizadas as UFs colocadas na Tabela 5-1. Esta tabela mostra o nome das UFs, a funcionalidade de cada uma, a tela em que o telefone precisa estar para que a UF seja invocada, a tela (ou seja, o eestado) em que o telefone estará depois da execução da UF, e os métodos disponíveis na API de cada UF.

A especificação XML listada no Anexo 4 foi escrita de acordo com Tabela 5-1. Utilizando a ferramenta produzida como resultado de pesquisa deste trabalho, cria-se de forma automatizada um aspecto que, quando compilado com o TAF, instancia uma rede de Petri que controla a chamada de métodos de cada UF e o estado do telefone. Como conseqüência, a ordem em que as UFs são colocadas em um caso de teste será verificada de acordo com a pré e pós-condição de cada UF que compõe o teste.

Nome da UF	Funcionalidade	Tela de pré-condição	Tela de pós-condição	Métodos da API
LaunchApp	Abre uma aplicação	Qualquer tela	Qualquer tela	- *execute()* – implementação do comportamento da UF - *setApplication(Phone Application)* – define a aplicação que será iniciada
CapturePicture FromCamera	Tira uma fotografia utilizando a câmera do telefone	*Photo viewfinder*	*Save discard screen*	- *execute()* – implementação do comportamento da UF
Store Multimedia FileAs	Toma uma ação com a fotografia recentemente tirada (como salvar ou discartar a fotografia, enviá-la por mensagem ou e-mail, etc)	*Save discard screen*	*Photo viewfinder, Video viewfinder, Browser application*	- *execute()* – implementação do comportamento da UF - *setStoreOption (MultimediaItem)* – define a ação a ser feita com a fotografia
ScrollToAnd Select MultimediaFile	Navega por uma lista de arquivos multimídia e seleciona o arquivo especificado	*Sounds file list, Pictures file list, Videos file list*	*Sounds file list, Pictures file list, Videos file list*	- *execute()* – implementação do comportamento da UF - *setItem(Multimedia File)* – define o arquivo a ser selecionado
CheckScreen	Verifica se o telefone encontra-se em uma tela específica	Nenhum	Nenhum	- *execute()* – implementação do comportamento da UF - *setTitle(Application Screen)* – define a tela em que o telefone deve estar no momento da chamada à esta UF
OpenCurrent FileDetails	Abre a janela de detalhes do arquivo correntemente selecionado ou aberto	*Sound player, Picture viewer, Video player*	*File details screen*	- *execute()* – implementação do comportamento da UF
VerifyAll Multimedia FileDetails	Verifica se cada campo de detalhes de um arquivo multimídia estão sendo corretamente mostrados	*File details screen*	*File details screen*	- *execute()* – implementação do comportamento da UF - *setFile(Multimedia File)* – define qual o arquivo que será verificado
ReturnTo PreviousScreen	Retorna para a tela anterior	Qualquer tela	Qualquer tela	- *execute()* – implementação do comportamento da UF
DeleteFile	Remove o arquivo multimídia selecionado	*Sounds file list, Pictures file list, Videos file list*	*Sounds file list, Pictures file list, Videos file list*	- *execute()* – implementação do comportamento da UF - *setFile(Multimedia File)* – define o arquivo que será apagado

Tabela 5-1: Algumas UFs do TAF com suas funcionalidades e restrições

A verificação de chamada de métodos da API das UFs é feita automaticamente com a evolução das marcações da rede de Petri à medida em que os métodos são chamados. Suponhamos, por exemplo, que a UF *LaunchApp* seja instanciada e adicionada à lista de *Steps* de um caso de teste, mas o método *setApplication* não foi invocado para passar à esta UF a aplicação para a qual a UF deve navegar. Nessa situação, observa-se claramente a violação da expressão regular que rege a chamada de métodos da API da UF *LaunchApp* especificada pelo elemento *methodsCallOrder* do arquivo XML do Anexo 4. O sistema de *log* do TAF sinaliza essa violação de restrição do *framework* conforme mostrado na Figura 5.5.

```
23-10-2005/01:31.14  Test case start: com.motorola.testcase.s004.TC_test (#1)
23-10-2005/01:31.15  Precondition: EMPTY:
23-10-2005/01:31.15  Procedure:
23-10-2005/01:31.15  navigationTk.launchApp(PhoneApplication.CAMERA);
23-10-2005/01:31.15
com.motorola.systemvalidation.petrinet.PetriNetTransitionNotAllowed
23-10-2005/01:31.15    at
com.motorola.taf.framework.aspects.TAFAspectValidation.ajc$before$com_motorola_taf_frame
work_aspects_TAFAspectValidation$18$c059e856(TAFAspectValidation.aj:759)
23-10-2005/01:31.15    at
com.motorola.taf.utility.function.navigation.p2k.LaunchAppImp.execute(LaunchAppImp.java:
-1)
23-10-2005/01:31.15  Test case end: FAILED in 00:00:01 :
com.motorola.testcase.integration.bas.s004.TC_test
```

Figura 5.5: Resultado de uma execução onde a ordem de métodos não é respeitada

Com relação à ordem em que as UFs estão dispostas no teste do Anexo 3, observa-se que esta está correta: a pré-condição de uma UF é cumprida pela pós-condição da UF anterior. Nesse caso, o resultado da execução do teste é positivo, conforme mostra o sistema de *log* do TAF na Figura 5.6.

```
18-10-2005/07:46.33  Test case start: com.motorola.testcase.s004.TC_test (#1)
18-10-2005/07:46.33     Procedure:
18-10-2005/07:46.33              navigationTk.launchApp(PhoneApplication.CAMERA);
18-10-2005/07:46.50              multimediaTk.capturePictureFromCamera();
18-10-2005/07:46.50
multimediaTk.storeMultimediaFileAs(MultimediaItem.STORE_ONLY);
18-10-2005/07:46.55              navigationTk.launchApp(PhoneApplication.PICTURES);
18-10-2005/07:47.05
multimediaTk.scrollToAndSelectMultimediaFile(MultimediaFile.UNNAMED);
18-10-2005/07:47.12              phoneTk.checkScreen(MultimediaScreen.MEDIA_PLAYER);
18-10-2005/07:47.12              multimediaTk.openCurrentFileDetails();
18-10-2005/07:47.12
multimediaTk.verifyAllMultimediaFileDetails(MultimediaFile.UNNAMED);
18-10-2005/07:47.12              phoneTk.returnToPreviousScreen();
18-10-2005/07:47.23              phoneTk.returnToPreviousScreen();
18-10-2005/07:47.32              multimediaTk.deleteFile(MultimediaFile.UNNAMED, true);
18-10-2005/07:47.37  Test case end: PASSED in 00:01:04 :
com.motorola.testcase.integration.bas.s004.TC_test
```

Figura 5.6: Resultado da execução bem-sucedida do caso de teste do Anexo 3

Se o caso de teste não estiver observando as restrições impostas pelas pré e pós-condições das UFs, o sistema de log sinalizará que o resultado da execução falhou como uma exceção *PetriNetTransitionNotAllowed*. O log da Figura 5.7 mostra o resultado da execução do teste do Anexo 3 quando é removida a marca de comentário "//" da primeira chamada à UF *DeleteFile*.

```
18-10-2005/07:32.27  Test case start: com.motorola.testcase.integration.TC_test (#1)
18-10-2005/07:32.27     Precondition: EMPTY:
18-10-2005/07:32.27     Procedure:
18-10-2005/07:32.27              navigationTk.launchApp(PhoneApplication.CAMERA);
18-10-2005/07:32.45              multimediaTk.capturePictureFromCamera();
18-10-2005/07:32.45
multimediaTk.storeMultimediaFileAs(MultimediaItem.STORE_ONLY);
18-10-2005/07:32.50              navigationTk.launchApp(PhoneApplication.PICTURES);
18-10-2005/07:32.58
multimediaTk.scrollToAndSelectMultimediaFile(MultimediaFile.UNNAMED);
18-10-2005/07:33.10              phoneTk.checkScreen(MultimediaScreen.MEDIA_PLAYER);
18-10-2005/07:33.10              multimediaTk.openCurrentFileDetails();
18-10-2005/07:33.10
multimediaTk.verifyAllMultimediaFileDetails(MultimediaFile.UNNAMED);
18-10-2005/07:33.10              multimediaTk.deleteFile(MultimediaFile.UNNAMED, true);
18-10-2005/07:33.10
com.motorola.systemvalidation.petrinet.PetriNetTransitionNotAllowed
18-10-2005/07:33.10    at
com.motorola.taf.framework.aspects.TAFAspectValidation.ajc$before$com_
motorola_taf_framework_aspects_TAFAspectValidation$69$c059e856(TAFAspe
ctValidation.aj:3284)
18-10-2005/07:33.10  Test case end: FAILED in 00:00:43 :
com.motorola.testcase.integration.bas.s004.TC_test
```

Figura 5.7: Log de um caso de teste que não observa as restrições das UFs

A versão modificada do teste do Anexo 3 falha porque a UF *DeleteFile* não pode ser chamada depois da UF *VerifyAllMultimediaFileDetails*. A pós-condição de *VerifyAllMultimediaFileDetails,* a tela *File details screen*, não é compatível com a pré-condição de *DeleteFile,* que é qualquer um dos estados representados pelas telas *Sounds file list*, *Pictures file list* ou *Videos file list*. Isto faz com que o resultado da execução do teste falhe, e o desenvolvedor do teste saberá facilmente onde o erro se encontra.

Existem UFs que não determinam especificamente em qual estado o telefone estará depois de sua execução (são os casos em que a pós-condição de um método é especificada com *postConditionAnyState* ou *postConditionAnyStateOf*). É o caso, por exemplo, de *LaunchApp* e *StoreMultimediaFileAs*. A rede de Petri neste caso possuirá lugares caracterizados com conflitos, assim como explanado na seção 4.4.4. Para resolver o conflito, o aspecto automaticamente gerado faz uso de um *Handler*, assim como ilustrado na seção 4.5.1 para o *framework* de sistemas bancários.

Implementou-se um *handler* especial para o TAF, de forma que o desenvolvedor que está implementando o teste pode escolher para qual estado o sistema vai no momento em que o conflito efetivo acontece. Na Figura 5.8 apresenta-se a tela do *TAF Validator Tool*, a ferramenta que implementa o *handler* para o TAF. Na situação da Figura 5.8, o usuário pode escolher para qual estado o telefone irá depois da execução da primeira chamada à UF *LaunchApp* do caso de teste do Anexo 3.

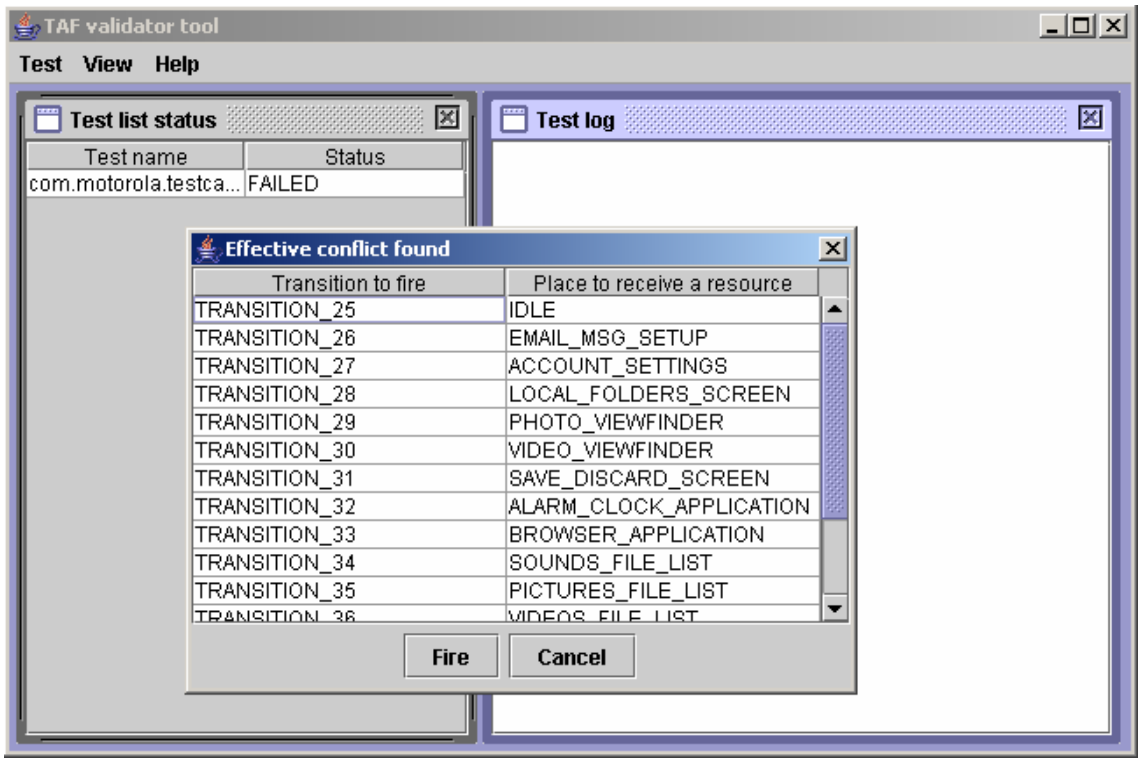


Figura 5.8: Tela do TAF Validator tool

5.5 Avaliação dos resultados obtidos pela especificação formal de restrições para o TAF

Conforme já mencionado, o TAF foi projetado para possibilitar a construção de casos de teste automatizados objetivando verificar se os requisitos funcionais de um telefone estão sendo corretamente implementados, isto é, verifica-se se o comportamento do telefone é realmente o esperado. Além de verificar as funcionalidades do telefone, a execução dos testes automatizados com o TAF também se propõe a encontrar defeitos na fase em que o *software* do telefone está sendo desenvolvido. Dito de outra forma, os casos de teste deverão verificar as

funcionalidades do telefone e, além disso, encontrar os pontos em que o aparelho celular apresenta falhas. Se a execução de um caso de teste for bem sucedida, supõe-se que a funcionalidade verificada por este teste está funcionando corretamente e o resultado deste teste será *PASSED*. Se alguma ação ou verificação requerida pelo teste não se não se comportar conforme o esperado, o resultado deste teste será *FAILED*. Vale salientar, entretanto, que a execução bem sucedida de um conjunto de testes de modo algum garante que o telefone esteja livre de defeitos.

O fato de o *software* do telefone celular estar sendo testado de forma automatizada por um outro *software* constitui um desafio: como saber se o *testware* (o *software* testador) está correto? Este é o dilema em que o desenvolvedor do *testware* se encontra, especialmente no momento em que um teste automatizado falha. Como saber se a falha está no *software* sob teste ou se a falha está no *testware*? Por outro lado, como saber se o telefone realmente se comporta conforme o esperado se o resultado do teste automatizado é positivo? Não estaria o teste automatizado deixando passar por alto alguma verificação importante?

Muito tem sido publicado pela comunidade científica internacional para solucionar os problemas acima expostos da maneira mais satisfatória possível. O fato principal a considerar na automação de testes é que desenvolve-se *software* para testar *software*. Se o *software* testador não for suficientemente confiável, tampouco o *software* testado o será.

A abordagem de especificação formal para o TAF apresentada neste trabalho certamente não soluciona por completo as principais questões envolvidas no caso específico da automação de testes, mas o *testware* TAF se torna mais confiável como ferramenta para automação. Menciona-se abaixo alguns pontos que justificam esta afirmação:

a) a API das UFs será utilizada corretamente no sentido de que os parâmetros necessários à execução de cada UF serão obrigatoriamente fornecidos; em outras palavras, todos os métodos *set* presentes nas APIs das UFs cuja finalidade seja passar às UFs os parâmetros indispensáveis à sua execução serão obrigatoriamente invocados;
b) os casos de teste construídos sob o TAF serão compostos de forma

que a seqüência de chamada das UFs é tal que as pré e pós-condições especificadas em termo da tela em que o telefone está serão obrigatoriamente respeitadas.

Como conseqüência dos pontos acima, pode-se enumerar os seguintes resultados da aplicação do método formal proposto neste trabalho no TAF:

a) *O tempo de desenvolvimento gasto para automatizar testes diminui*. Na fase de implementação do caso de teste, a única forma que o desenvolvedor tem para se assegurar de que o caso de teste está correto é executá-lo e esperar que o teste passe. Isto, entretanto, pode ser caro: os casos de teste não raro são longos e consomem muito tempo para serem executados. É frustante para o desenvolvedor observar que o teste falhou nos últimos passos (depois de esperar pelo menos uns 20 minutos desde o início da execução do teste automatizado) porque uma UF foi chamada no lugar indevido. Muitas vezes, isso também não é fácil constatar, o que obriga o desenvolvedor a depurar boa parte do código do teste até descobrir o ponto aonde está o erro.
b) *O tempo de manutenção dos testes automatizados diminui*. Os casos de teste que não são eficientes em encontrar defeitos no telefone são rapidamente alterados ou removidos (afinal, um dos objetivos do teste é encontrar as falhas). No caso em que um teste é alterado, a inclusão, exclusão ou mudança de passos poderá tornar a seqüência de UFs do teste automatizado incorreta. Quaisquer inconsistências deste tipo serão rapidamente detectadas pelo esquema formal proposto neste trabalho, e o desenvolvedor poderá corrigi-las prontamente.
c) *O código das implementações concretas das UFs torna-se menor e mais legível*. Não é necessária a codificação de rotinas de verificação na implementação das UFs. A inserção de sentenças *if* que verificam se certos atributos do objeto são objetos não-nulos (isto é, sentenças que verificam se um determinado método *set* foi invocado antes da

execução da UF) torna-se redundante quando utilizada a especificação formal das UFs proposta neste trabalho.

d) *A verificação de ordem de chamada de métodos aplica-se automaticamente a qualquer implementação das UFs formalmente especificadas*. Uma UF pode ter várias implementações, criadas para lidar com telefones diferentes. Neste caso, a especificação formal desenvolvida neste trabalho evita que as sentenças de verificação de ordem de chamada de métodos sejam repetidas em diferentes implementações de uma mesma UF. Isto é alcançado, pois as classes que implementam concretamente as APIs abstratas das UFs herdam os aspectos compilados com as APIs.

Para o desenvolvedor, o custo pago para alcançar os benefícios da especificação formal segundo a abordagem sugerida nesse trabalho é equivalente ao trabalho realizado para especificar as UFs no formato XML, segundo o esquema do Anexo 1. Considera-se, entretanto, que o custo pago para produzir esta especificação não é realmente grande, uma vez que o tempo que o desenvolvedor deixará de perder para descobrir que uma chamada de método foi colocada no lugar errado ou que um caso de teste foi escrito de maneira incorreta é significativo.

Outro fator positivo a considerar pela escolha da especificaçao formal é a redução do esforço gasto para documentar o TAF. Gasta-se tempo para produzir documentação que seja útil para que os usuários do TAF o possam utilizar de forma correta. Se parte desta documentação for substituída pela descrição formal do *framework*, o projetista do TAF será recompensado com a verificação automática da correta ordem de chamada de métodos. Sob esse ponto de vista, o esforço para extender o TAF para o suporte de novas UFs não é tão maior quando estiver usando a especificação formal, além do fato de que o desenvolvedor não perderá tempo para depurar código que é automaticamente verificado.

Conclui-se que a aplicação da abordagem proposta ao TAF permitiu uma alteração significativa na lógica de produção e validação de casos de teste. Da forma como o TAF vinha sendo usado, a avaliação do *software* de teste ocorria executando-o junto com o *software* a ser testado. Assim, não seria claro a *priori* se uma falha na

execução correspondia a um erro no *software* sob teste ou no *software* de teste – apenas um processo de depuração levaria a um diagnóstico preciso. A partir da aplicação da abordagem proposta, há uma clara separação entre a validação do *software* de teste e do *software* sob teste. Uma inconsistência na definição de um caso de teste é detectada e automaticamente indicada, levando a uma depuração mais rápida do *software* de teste, e que ocorre em um momento diferente daquele da avaliação do *software* sob teste.

6 CONCLUSÃO

Como conclusão geral da presente pesquisa, observa-se que os métodos formais são valiosos para a validação de sistemas construídos sob *frameworks* orientados a objetos. Observou-se que sistemas construídos sob *frameworks* com restrições especificadas formalmente tendem a ser desenvolvidos em menos tempo, além de serem mais confiáveis e apresentarem menos defeitos, em função da indicação automatizada de eventuais inconsistências – em contraste com um processo tradional de depuração, caso a indicação de inconsistências não ocorresse automaticamente.

Constata-se também nessa pesquisa que a geração automática de código proporcionada pelo método de descrição formal proposto nesse trabalho é vantajoso por pelo menos três razões: o desenvolvedor do *framework* será poupado da tarefa massante e dispendiosa de escrever (manualmente) código de verificação; parte da documentação do *framework* pode ser escrita de maneira formal, tendo como ganho a verificação automática das restrições especificadas; durante a fase de testes, os erros referentes a mau uso do *framework* serão mais facilmente encontrados e diagnosticados.

A seguir, apresentam-se: os principais resultados obtidos por essa pesquisa, isto é, a forma como os objetivos do trabalho foram alcançados; limitações; sugestões para pesquisas e trabalhos futuros; e algumas considerações finais.

6.1 Resultados obtidos

Como já dito anteriormente, o objetivo deste trabalho centraliza-se em definir formalmente restrições que um *framework* possui em termos da ordem em que os métodos de objetos do *framework* podem ser invocados. Para alcançar este objetivo, foi criado um esquema XML para reger a escrita de uma especificação que possibilitasse a geração automática de código capaz de impedir a chamada de métodos em ordem incorreta.

A especificação formal XML de restrições de um *framework* e a geração automática de código produzido pela ferramenta obtida como resultado desta pesquisa, possibilitaram o alcance dos seguintes objetivos mencionados no capítulo introdutório:

a) O uso de XML facilita a especificação formal, uma vez que esta notação tem sido bastante difundida e utilizada em aplicações

desenvolvidas para a Internet, como meio de integrar e possibilitar a comunicação entre sistemas diferentes, pelos Serviços Web (*Web Services*) (SUN, 1994) como protocolo de comunicação entre processos, etc. A disseminação de XML, portanto, constitui um fator positivo para facilitar a escrita da especificação formal por programadores que não são especialistas em formalizar sistemas através de métodos como máquinas de estado ou redes de Petri. Outro ponto importante a mencionar é que a estrutura de elementos aninhados na qual XML é fundamentada é equivalente às contruções de uma expressão regular; intuitivamente, o programador conhecedor de XML especificará o *framework* sem se dar conta de que, na verdade, ele está escrevendo uma expressão regular que dará origem a uma máquina de estados, e que esta máquina, por sua vez, será utilizada para montar uma rede de Petri. O processo de montagem da rede de Petri acontece de forma absolutamente transparente ao programador desenvolvedor do *framework*.

b) Conforme apresentado no Capítulo 4, gerou-se automaticamente o aspecto listado no Anexo 2 para o *framework* de sistemas bancários. Quando compilado junto com o código da aplicação construída sob o *framework*, esse aspecto é capaz de validar em tempo de execução as chamadas a métodos dos objetos do *framework*. Visto que o *framework* de sistemas bancários, objeto de estudo do capítulo 4, trata-se de um exemplo fictício, o TAF, um *framework* real utilizado no meio corporativo, foi utilizado como estudo de caso, e os resultados da abordagem proposta nesse trabalho para o TAF foram apresentados no Capítulo 5.

c) O mau uso de um *framework* pode acarretar em perda de tempo de desenvolvimento e em atrasos no cumprimento de prazos e na entrega do produto final ao cliente em função de necessidade de um processo de depuração para buscar e corrigir problemas. Atrasos, sem dúvida, geram insatisfação por parte do cliente e em *déficits* no orçamento do projeto como um todo. A especificação formal de

restrições, segundo a abordagem proposta nesse trabalho possibilita ao desenvolvedor da aplicação construída sob o *framework* a rápida detecção de qualquer problema relacionado a uma chamada de método indevida, minimizando o esforço gasto para a correção do problema. No caso do TAF, apresentado no Capítulo 5, mostrou como o ganho de tempo em depurar casos de teste pode ser significativo.

d) Defeitos que escapam na fase de teste de um *software* e que são detectados pelo cliente são críticos. Dependendo da aplicação, as conseqüências podem ser desastrosas e os prejuízos podem ser significativos. Conforme mostrado nos Capítulos 4 e 5, as chances de escaparem defeitos decorrentes do mau uso do *framework* no sentido da invocação incorreta de métodos se tornam menores, com a aplicação da abordagem proposta.

e) O controle dos métodos invocados é feito a partir de uma rede de Petri, a qual evolui o estado de sua marcação na medida em que o sistema é executado e os métodos do *framework* são invocados. Essa rede de Petri também é dinâmica no sentido de que lugares e transições são adicionados à rede no momento em que novos objetos são criados ou destruídos. Essa capacidade possibilita o completo controle do estado do sistema.

6.2 Limitações

6.2.1 Limitações do modelo

O modelo de especificação formal proposto neste trabalho limita-se a especificar a ordem de chamada aos métodos dos objetos que um *framework* provê. Essa, entretanto, não é a única classe de restrições que um *framework* pode impor a seus usuários. Outras questões que precisam ser observadas quando um *framework* estiver sendo usado incluem: implementar todos os *hot spots* obrigatórios; utilização dos métodos permitidos somente (HOU & HOOVER, 2001); verificação de asserções de baixo nível (estruturas de dados e variáveis locais) (UBAYASHI & TETSUO, 2002), a

sobrescrição apenas dos métodos permitidos (SILVA & FRIEBERGER 2004). O modelo formal proposto nesse trabalho não suporta a especificação de restrições associadas a essas questões.

A ferramenta produzida como resultado desta pesquisa gera automaticamente um aspecto para ser usado primariamente como ferramenta para auxiliar a fase de testes e depuração de uma aplicação construída sob um *framework*. Quando utilizado em conjunto com a fase de testes, a verificação do estado do sistema e do estado dos objetos em tempo de execução sinalizará qualquer violação de ordem de chamada de métodos. Essa abordagem, entretanto, possui uma limitação: apenas os caminhos de execução previstos pelos testes do sistema são cobertos. Os trabalhos publicados por HOU & HOOVER (2001), UBAYASHI & TETSUO (2002) e NASA (1999) tratam meios para contornar essa limitação.

O modelo também não prevê a modelagem de sistemas concorrentes ou paralelos, embora possa ser estendido para contemplá-los. Nenhuma experiência ainda foi feita com um *framework* que utiliza várias *threads* de execução.

Ainda outra limitação do modelo, conforme explanado no capítulo 4, seção 4.5.1, é o fato de que a especificação XML das restrições do *framework* se atém às classes que o compõem, e não aos objetos. Isto implica que o modelo não pode monitorar múltiplas instâncias de objetos de uma mesma classe ao mesmo tempo (isto é, os objetos transientes que são criados e destruídos em tempo de execução).

6.2.2 Limitações da ferramenta de geração de aspectos

Uma característica do aspecto automaticamente gerado pela ferramenta que é resultado desta pesquisa é o fato de que o disparo de transições da rede de Petri é feito no momento em que os métodos são realmente executados. Não há atualmente nenhuma maneira automatizada de validar a rede de Petri que é montada para um *framework* específico. Pode haver, por exemplo, *deadlocks* ou transições que nunca estarão habilitadas para serem disparadas. Um *deadlock* pode ocorrer, por exemplo, se a pré e pós condição de um método incluírem o mesmo estado. A Figura 6.1 mostra uma rede de Petri com *deadlock*.

A Rede da Figura 6.1 é uma versão modificada da Rede da Figura 4.18. O problema dessa rede está na pré e pós-condição do método *auth(Credential)*: ambos

incluem o estado AUTHENTICATED. O *deadlock* nesse caso caracteriza-se pelo arco com linha descontínua: visto que o lugar AUTHENTICATED *nunca* receberá uma ficha, a transição *auth(Credential)* nunca estará habilitada.

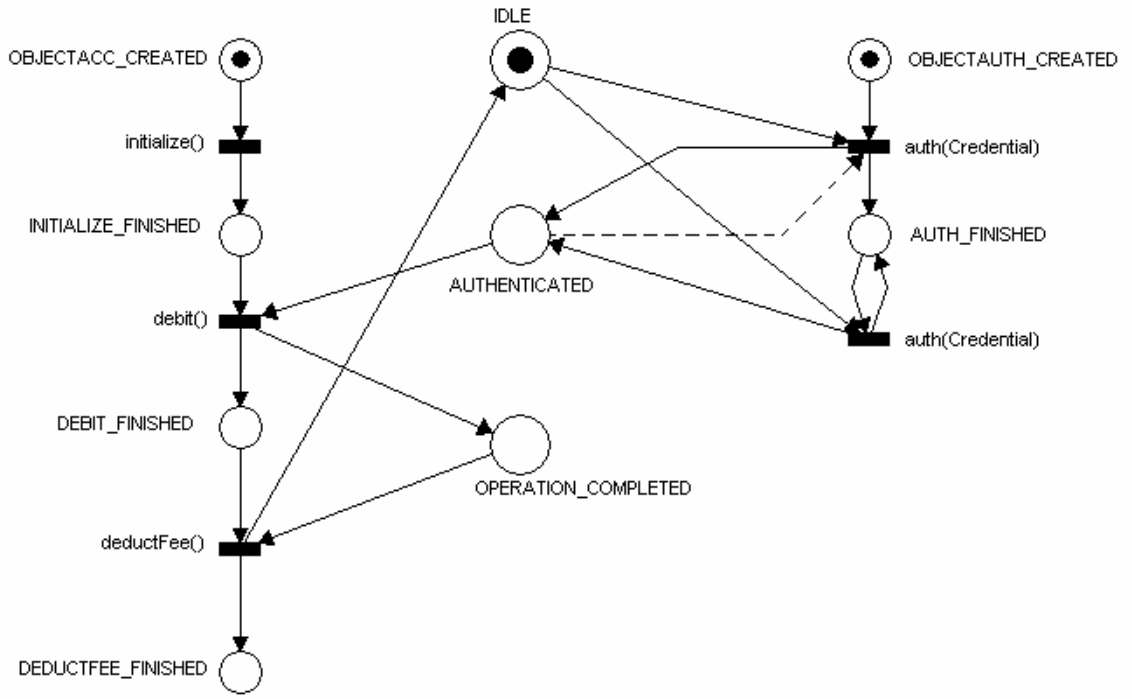


Figura 6.1: *Deadlock* em uma rede de Petri com inconsistência em sua especificação

Se a rede de Petri possuir qualquer uma das inconsistências acima mencionadas, o aspecto automaticamente gerado poderá gerar uma exceção em tempo de execução dizendo que um método foi incorretamente invocado, o que realmente não é verdade: a rede de Petri é que não está corretamente constituída. Isso poderá confundir o desenvolvedor, pois o sistema construído sob o *framework* está correto, mas a sua especificação formal é inconsistente.

Outra limitação da ferramenta é a falta de uma interface amigável ao desenvolvedor do *framework* para escrever a especificação formal XML. A especificação XML exige que todas as classes e métodos mencionados na especificação sejam completamente qualificados (isto é, as classes deverão incluir o nome do pacote completo, os métodos deverão incluir modificadores como *public ou protected*, além do tipo de dado de retorno dos métodos). Isso dificulta o processo de especificação e pode desencorajar o desenvolvedor do *framework* a utilizar o método formal proposto nesta

pesquisa.

6.3 Trabalhos futuros

6.3.1 Validação da especificação formal

Para resolver o problema da verificação de inconsistência da especificação (que pode gerar uma rede de Petri com *deadlocks* ou estados inalcançáveis) sugerem-se duas abordagens. A primeira seria analisar a rede de Petri gerada pela especificação formal. Para isso, pode-se utilizar a abordagem de análise de propriedades de redes de Petri utilizadas por CUNHA (2005). A segunda forma seria analisar a própria especificação XML e evitar que uma rede de Petri inconsistente seja gerada.

6.3.2 Suporte para sistemas concorrentes

Na rede de Petri automaticamente gerada, cria-se uma classe conforme o padrão de projeto *Singleton* para conter os lugares e transições que representam os estados do sistema. Esta classe *Singleton* é única, e é mantida em memória durante todo o tempo em que o sistema está sendo executado. Por outro lado, são criados objetos para representar os lugares e transições que controlam os estados dos objetos transitórios (os objetos que são criados e destruídos em tempo de execução).

A classe *Singleton* que controla os estados do sistema atualmente implementada considera que existe apenas uma *thread* de execução. Com pouco esforço, pode-se implementar um monitor para esta classe, de forma que ela permita o acesso de somente uma *thread* de execução por vez. Desta forma, as fichas colocadas nos lugares que representam o estado do sistema poderiam representar recursos pelos quais os processos concorrentes poderiam estar disputando entre si.

Outro ponto necessário para fazer a atual implementação suportar sistemas com múltiplas *threads,* seria alterar o aspecto automaticamente gerado para tirar as fichas dos lugares de entrada das transições antes do início da execução do método, e colocar as fichas nos lugares de saída somente depois do término da execução do referido método. A implementação atual faz ambas operações antes do início de cada método.

Sugere-se, portanto, como tema de pesquisa futura, verificar em quais casos o

método formal sugerido neste trabalho se aplica para especificar sistemas concorrentes. Se for levantado algum caso onde o esquema XML proposto não é apropriado para a especificação de sistemas concorrentes, poderiam ser propostas extensões do esquema para estes casos.

6.3.3 Utilização de Redes de Petri de Alto Nível

Conforme limitação do modelo exposta na seção 6.2.1, e no capítulo 4, seção 4.5.1, referente ao monitoramento de múltiplas instâncias simultâneas de objetos de classes que compõem o *framework*, sugere-se o estudo de Redes de Petri de Alto Nível (CARDOSO & VALETTE, 1997) para a solução deste problema. Redes de Petri Coloridas possivelmente podem ser usadas para criar a correspondência entre instâncias de classes diferentes.

6.3.4 Geração de Redes de Petri em formato padronizado

Conforme Figura 4.1, a saída gerada pela ferramenta geradora de aspectos é um código orientado a aspectos escrito em *AspectJ*. Essa ferramenta poderia, entretanto, gerar também uma Rede de Petri escrita em um formato padrão, como, por exemplo, a PNML – *Petri Net Markup Language* (JÜNGEL et al, 2000). A principal motivação para a geração de uma rede em formato padrão é a reutilização que a padronização proporciona. Uma rede de Petri construída em PNML poderia ser lida e eventualmente reutilizada por qualquer ferramenta que suporte o PNML.

6.3.5 Suporte para auxílio à especificação formal

Conforme destacado na seção de limitações, o formato de entrada XML para especificação do *framework* pode tornar a escrita da especificação um tanto trabalhosa. Uma ferramenta visual para auxiliar o desenvolvedor do *framework* a escrever essa especificação certamente diminuiria o tempo gasto com a especificação formal. Basicamente, uma ferramenta como essa deveria possibilitar ao usuário entrar com os estados do sistema, definir a ordem de chamada de métodos das classes necessárias, e entrar com as pré e pós-condições de cada método. Sugere-se criar um navegador que auxilia o usuário a escolher as classes e os seus métodos de maneira visual para

"montar" o arquivo XML. Um *plug-in* para o ECLIPSE (2005), por exemplo, poderia ser implementado para isso com relativamente pouco esforço. O protótipo da *interface* com o usuário deste *plug-in* é mostrado na Figura 6.2.

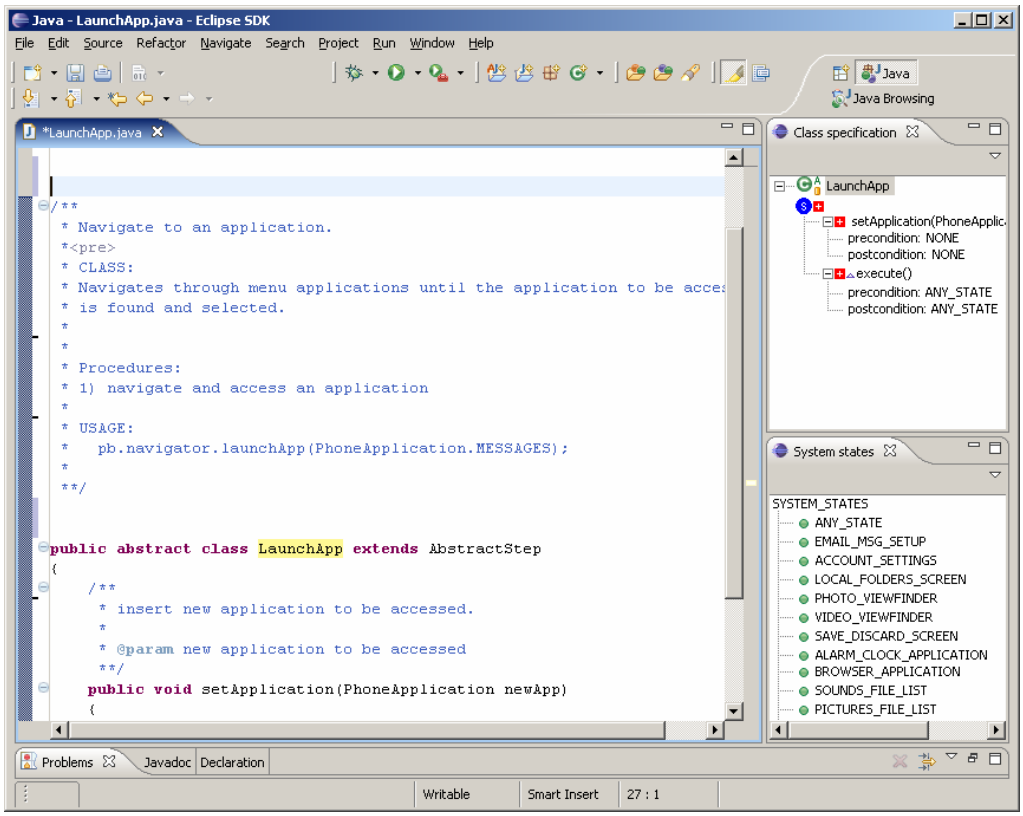


Figura 6.2: Protótipo da interface do usuário de um plug-in para auxiliar a especificação formal

6.4 Considerações finais

Os *frameworks* orientados a objetos são poderosas ferramentas para promover o reuso de projeto e código e o desenvolvimento rápido de aplicações em um determinado domínio. Desenvolver um *framework*, entretanto, é um processo mais complexo que desenvolver uma aplicação específica, pois está envolvido o entendimento completo e a generalização de um domínio de aplicações. Uma vez construído o *framework*, há o desafio de saber usá-lo corretamente (evitar o seu mau uso), o que em muitos casos, pode evitar conseqüências desastrosas. Os métodos formais, além de serem extensivamente usados na academia, podem ser aplicados com sucesso para aplicações do mundo real e, preferencialmente, sem demandar aumento significativo de esforço, tanto do desenvolvedor de um *framework* quanto do

desenvolvedor da aplicação construída sob um *framework*, poupando a ambos de tarefas dispendiosas de depuração e facilitando a correção de erros. O presente trabalho além de se preocupar com a construção de uma solução que aplicasse formalismo para evitar o mau uso de *frameworks*, também priorizou que essa introdução de formalismo não implicasse em aumento de esforço tal que comprometesse sua aplicabilidade. Espera-se com isso ter produzido uma contribuição que auxilie a difusão da abordagem de desenvolvimento de software baseado em *frameworks* orientados a objetos.

ANEXO 1: ESQUEMA XML PARA ESPECIFICAÇÃO FORMAL DE INTERFACE DE CLASSES

Arquivo: system_schema.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:cs="http://www.motorola.com/systemspecification"
    xmlns:ct="http://www.motorola.com/systemspecification/commontypes"
    targetNamespace="http://www.motorola.com/systemspecification"
    elementFormDefault="qualified">

    <import namespace="http://www.w3.org/1999/xlink"
schemaLocation="xlink.xsd"/>
    <import
namespace="http://www.motorola.com/systemspecification/commontypes"
schemaLocation="common_types.xsd"/>

    <element name="systemSpecification">
        <complexType>
            <sequence>
                <element name="systemStates"
type="cs:systemStatesType" minOccurs="0" maxOccurs="1"/>
                <element name="class" type="cs:classType"
minOccurs="1" maxOccurs="unbounded"/>
            </sequence>
            <attribute ref="xlink:type" use="required"
fixed="extended"/>
        </complexType>
    </element>

    <complexType name="systemStatesType">
        <sequence>
            <element name="state" minOccurs="1" maxOccurs="unbounded">
                <complexType>
                    <attribute ref="xlink:label" use="required"/>
                    <attribute name="initial" type="ct:booleanYesNo"
use="optional" default="no"/>
                </complexType>
            </element>
        </sequence>
    </complexType>

    <complexType name="classType">
        <sequence>
            <element name="methodsRestrictions" minOccurs="0"
maxOccurs="1">
                <complexType>
                    <sequence>
                        <element name="method"
type="cs:methodRestrictionType" minOccurs="1" maxOccurs="unbounded"/>
                    </sequence>
                </complexType>
```

```
            </element>
            <element name="methodsCallOrder" minOccurs="0"
maxOccurs="1">
                <complexType>
                    <sequence>
                        <element name="list" type="cs:methodTreeType"
minOccurs="1" maxOccurs="1"/>
                    </sequence>
                </complexType>
            </element>
        </sequence>
        <attribute name="name" type="ct:javaType" use="required"/>
    </complexType>

    <complexType name="methodTreeType">
        <sequence>
            <choice minOccurs="1" maxOccurs="unbounded">
                <element name="methodCall" type="cs:methodCallType"/>
                <element name="list" type="cs:methodTreeType"/>
            </choice>
        </sequence>
        <attribute name="type" type="cs:listTypeType" use="required"/>
        <attribute name="quantifier" type="cs:quantifierType"
use="optional" default="1"/>
    </complexType>

    <simpleType name="listTypeType">
        <restriction base="NMTOKEN">
            <enumeration value="sequence"/>
            <enumeration value="choice"/>
        </restriction>
    </simpleType>

    <simpleType name="quantifierType">
        <restriction base="string">
            <enumeration value="+"/>
            <enumeration value="*"/>
            <enumeration value="?"/>
            <enumeration value="1"/>
        </restriction>
    </simpleType>

    <complexType name="methodCallType">
        <attribute name="signature" type="ct:stringNotEmpty"
use="required"/>
        <attribute name="quantifier" type="cs:quantifierType"
use="optional" default="1"/>
    </complexType>

    <complexType name="methodRestrictionType">
        <sequence>
            <choice minOccurs="0" maxOccurs="unbounded">
                <element name="preConditionState">
                    <complexType>
                        <sequence>
                            <element name="state" type="cs:stateType"
minOccurs="1" maxOccurs="1"/>
                        </sequence>
                    </complexType>
```

```
                </element>
                <element name="preConditionAnyStateOf">
                    <complexType>
                        <sequence>
                            <element name="state" type="cs:stateType"
minOccurs="2" maxOccurs="unbounded"/>
                        </sequence>
                    </complexType>
                </element>
                <element name="preConditionAnyState"/>
            </choice>
            <choice minOccurs="0" maxOccurs="unbounded">
                <element name="postConditionState">
                    <complexType>
                        <sequence>
                            <element name="state" type="cs:stateType"
minOccurs="1" maxOccurs="1"/>
                        </sequence>
                    </complexType>
                </element>
                <element name="postConditionAnyStateOf">
                    <complexType>
                        <sequence>
                            <element name="state" type="cs:stateType"
minOccurs="2" maxOccurs="unbounded"/>
                        </sequence>
                    </complexType>
                </element>
                <element name="postConditionAnyState"/>
            </choice>
        </sequence>
        <attribute name="signature" type="ct:stringNotEmpty"
use="required"/>
    </complexType>

    <complexType name="stateType">
        <attribute ref="xlink:to" use="required"/>
    </complexType>

</schema>
```

Arquivo: common_types.xsd

```
<?xml version="1.0" encoding="UTF-8"?>

<schema xmlns="http://www.w3.org/2001/XMLSchema"

xmlns:ct="http://www.motorola.com/compspecification/commontypes"

targetNamespace="http://www.motorola.com/systemspecification/commontyp
es"
        elementFormDefault="qualified">

    <!--
    An identifier is an unlimited-length sequence of "Java letters"
and
    "Java digits", the first of which must be a "Java letter".
    A "Java letter" includes lowercase and uppercase letters a-z, A-Z,
    dollar ($) and underscore (_).
```

```
    A "Java digit" includes the digits 0-9.
    -->
    <simpleType name="javaIdentifier">
        <restriction base="string">
            <pattern value="([A-Z]|[a-z]|[_$])([A-Z]|[a-z]|[0-
9]|[_$])*" />
        </restriction>
    </simpleType>

    <!--
    Java type is a list of concatened identifiers, separated by ".".
    -->
    <simpleType name="javaType">
        <restriction base="string">
            <pattern value="([A-Z]|[a-z]|[_$])([A-Z]|[a-z]|[0-
9]|[_$]|(\.([A-Z]|[a-z]|[_$])))*" />
        </restriction>
    </simpleType>

    <simpleType name="stringNotEmpty">
        <restriction base="string">
            <minLength value="1"/>
        </restriction>
    </simpleType>

    <simpleType name="booleanYesNo">
        <restriction base="NMTOKEN">
            <enumeration value="yes"/>
            <enumeration value="no"/>
        </restriction>
    </simpleType>

</schema>
```

Arquivo: xlink.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    elementFormDefault="qualified"
    targetNamespace="http://www.w3.org/1999/xlink">

    <attribute name="type">
        <simpleType>
            <restriction base="NMTOKEN">
                <enumeration value="simple"/>
                <enumeration value="extended"/>
                <enumeration value="locator"/>
                <enumeration value="arc"/>
                <enumeration value="resource"/>
                <enumeration value="title"/>
                <enumeration value="none"/>
            </restriction>
        </simpleType>
    </attribute>

    <attribute name="show">
        <simpleType>
```

```
            <restriction base="NMTOKEN">
                <enumeration value="new"/>
                <enumeration value="replace"/>
                <enumeration value="embed"/>
                <enumeration value="other"/>
                <enumeration value="none"/>
            </restriction>
        </simpleType>
    </attribute>

    <attribute name="actuate">
        <simpleType>
            <restriction base="NMTOKEN">
                <enumeration value="onLoad"/>
                <enumeration value="onRequest"/>
                <enumeration value="other"/>
                <enumeration value="none"/>
            </restriction>
        </simpleType>
    </attribute>

    <attribute name="label">
        <simpleType>
            <restriction base="NMTOKEN">
            </restriction>
        </simpleType>
    </attribute>

    <attribute name="from">
        <simpleType>
            <restriction base="NMTOKEN">
            </restriction>
        </simpleType>
    </attribute>

    <attribute name="to">
        <simpleType>
            <restriction base="NMTOKEN">
            </restriction>
        </simpleType>
    </attribute>

</schema>
```

ANEXO 2: ASPECTO AUTOMATICAMENTE GERADO PARA O *FRAMEWORK* DE SISTEMAS BANCÁRIOS

Arquivo: ATMState.java

```
// This file was automatically created. DO NOT CHANGE IT.
package com.fakebank.frmvalidation;

import com.motorola.compsvalidation.petrinet.Place;
import com.motorola.compsvalidation.petrinet.Transition;

public class ATMState
{
    /**
     * PRE_CONDITION_ANY_STATE refers to any of the following states:
     *    {IDLE, AUTHENTICATED, OPERATION_COMPLETED}
     * POST_CONDITION_ANY_STATE refers to any of the following states:
     *    {IDLE, AUTHENTICATED, OPERATION_COMPLETED}
     * POST_CONDITION_ANY_STATE_OF_1 refers to any of the following
     *    states:
     *    {IDLE, AUTHENTICATED}
     */
    public static final Place IDLE;
    public static final Place AUTHENTICATED;
    public static final Place OPERATION_COMPLETED;
    public static final Place PRE_CONDITION_ANY_STATE;
    public static final Place POST_CONDITION_ANY_STATE;
    public static final Place POST_CONDITION_ANY_STATE_OF_1;

    public static final Transition TRANSITION_1;
    public static final Transition TRANSITION_2;
    public static final Transition TRANSITION_3;
    public static final Transition TRANSITION_4;
    public static final Transition TRANSITION_5;
    public static final Transition TRANSITION_6;
    public static final Transition TRANSITION_7;
    public static final Transition TRANSITION_8;

    static
    {
        IDLE = new Place("IDLE");
        AUTHENTICATED = new Place("AUTHENTICATED");
        OPERATION_COMPLETED = new Place("OPERATION_COMPLETED");
        PRE_CONDITION_ANY_STATE = new
Place("PRE_CONDITION_ANY_STATE");
        POST_CONDITION_ANY_STATE = new
Place("POST_CONDITION_ANY_STATE");
        POST_CONDITION_ANY_STATE_OF_1 = new
Place("POST_CONDITION_ANY_STATE_OF_1");

        TRANSITION_1 = new Transition("TRANSITION_1");
        TRANSITION_2 = new Transition("TRANSITION_2");
        TRANSITION_3 = new Transition("TRANSITION_3");
        TRANSITION_4 = new Transition("TRANSITION_4");
        TRANSITION_5 = new Transition("TRANSITION_5");
```

```
        TRANSITION_6 = new Transition("TRANSITION_6");
        TRANSITION_7 = new Transition("TRANSITION_7");
        TRANSITION_8 = new Transition("TRANSITION_8");

        TRANSITION_1.addArcsFromPlace(IDLE, 1);
        TRANSITION_1.addArcsToPlace(PRE_CONDITION_ANY_STATE, 1);
        TRANSITION_2.addArcsFromPlace(AUTHENTICATED, 1);
        TRANSITION_2.addArcsToPlace(PRE_CONDITION_ANY_STATE, 1);
        TRANSITION_3.addArcsFromPlace(OPERATION_COMPLETED, 1);
        TRANSITION_3.addArcsToPlace(PRE_CONDITION_ANY_STATE, 1);
        TRANSITION_4.addArcsFromPlace(POST_CONDITION_ANY_STATE, 1);
        TRANSITION_4.addArcsToPlace(IDLE, 1);
        TRANSITION_5.addArcsFromPlace(POST_CONDITION_ANY_STATE, 1);
        TRANSITION_5.addArcsToPlace(AUTHENTICATED, 1);
        TRANSITION_6.addArcsFromPlace(POST_CONDITION_ANY_STATE, 1);
        TRANSITION_6.addArcsToPlace(OPERATION_COMPLETED, 1);
        TRANSITION_7.addArcsFromPlace(POST_CONDITION_ANY_STATE_OF_1,
1);
        TRANSITION_7.addArcsToPlace(IDLE, 1);
        TRANSITION_8.addArcsFromPlace(POST_CONDITION_ANY_STATE_OF_1,
1);
        TRANSITION_8.addArcsToPlace(AUTHENTICATED, 1);

        IDLE.setNoofResources(1);

    }
}
```

Arquivo: ATMAspectValidation.aj

```
// This file was automatically created. DO NOT CHANGE IT.

package com.motorola.taf.framework.aspects;

import com.motorola.compsvalidation.petrinet.*;
import com.motorola.compsvalidation.validationuserinteraction.*;

public aspect TAFAspectValidation
{
    AbstractIndeterminationHandler indetermHandler =
com.fakebank.ATM.validator.IndeterminationHandler.getInstance();
    /*
     * Aspect for component Account
     */
    private Place Account.STATE_1;
    private Place Account.STATE_2;
    private Place Account.STATE_3;
    private Place Account.STATE_4;

    private Transition Account.TRANSITION_9;
    private Transition Account.TRANSITION_10;
    private Transition Account.TRANSITION_11;

    private void Account.initializePetriNet()
    {
        STATE_1 = new Place("STATE_1");
        STATE_2 = new Place("STATE_2");
        STATE_3 = new Place("STATE_3");
        STATE_4 = new Place("STATE_4");
```

```
        TRANSITION_9 = new Transition("TRANSITION_9");
        TRANSITION_10 = new Transition("TRANSITION_10");
        TRANSITION_11 = new Transition("TRANSITION_11");

        TRANSITION_9.addArcsFromPlace(STATE_1, 1);
        TRANSITION_9.addArcsToPlace(STATE_2, 1);
        TRANSITION_10.addArcsFromPlace(STATE_2, 1);
        TRANSITION_10.addArcsToPlace(STATE_3, 1);

TRANSITION_10.addArcsFromPlace(com.fakebank.frmvalidation.ATMState.AUT
HENTICATED, 1);

TRANSITION_10.addArcsToPlace(com.fakebank.frmvalidation.ATMState.OPERA
TION_COMPLETED, 1);
        TRANSITION_11.addArcsFromPlace(STATE_3, 1);
        TRANSITION_11.addArcsToPlace(STATE_4, 1);

TRANSITION_11.addArcsFromPlace(com.fakebank.frmvalidation.ATMState.OPE
RATION_COMPLETED, 1);

TRANSITION_11.addArcsToPlace(com.fakebank.frmvalidation.ATMState.IDLE,
1);

        STATE_1.setNoofResources(1);

    }

    after(Account componentInstance):
        target(componentInstance) &&
        initialization(Account.new())
    {
        componentInstance.initializePetriNet();
    }

    before(Account componentInstance):
        target(componentInstance) &&
        execution(public void initializeAccount())
    {
        if (componentInstance.TRANSITION_9.enabled())
        {
            componentInstance.TRANSITION_9.fire();
        }
        else
        {
            throw new RuntimeException(new
com.motorola.compsvalidation.petrinet.PetriNetTransitionNotAllowed());
        }

    }

    before(Account componentInstance):
        target(componentInstance) &&
        execution(public void debit())
    {
        if (componentInstance.TRANSITION_10.enabled())
        {
            componentInstance.TRANSITION_10.fire();
        }
```

```
        else
        {
            throw new RuntimeException(new
com.motorola.compsvalidation.petrinet.PetriNetTransitionNotAllowed());
        }

    }

    before(Account componentInstance):
        target(componentInstance) &&
        execution(public void deductFee())
    {
        if (componentInstance.TRANSITION_11.enabled())
        {
            componentInstance.TRANSITION_11.fire();
        }
        else
        {
            throw new RuntimeException(new
com.motorola.compsvalidation.petrinet.PetriNetTransitionNotAllowed());
        }

    }

    /*
     * Aspect for component Authenticator
     */
    private Place Authenticator.STATE_1;
    private Place Authenticator.STATE_2;

    private Transition Authenticator.TRANSITION_12;
    private Transition Authenticator.TRANSITION_13;

    private void Authenticator.initializePetriNet()
    {
        STATE_1 = new Place("STATE_1");
        STATE_2 = new Place("STATE_2");

        TRANSITION_12 = new Transition("TRANSITION_12");
        TRANSITION_13 = new Transition("TRANSITION_13");

        TRANSITION_12.addArcsFromPlace(STATE_1, 1);
        TRANSITION_12.addArcsToPlace(STATE_2, 1);

TRANSITION_12.addArcsFromPlace(com.fakebank.frmvalidation.ATMState.IDL
E, 1);

TRANSITION_12.addArcsToPlace(com.fakebank.frmvalidation.ATMState.POST_
CONDITION_ANY_STATE_OF_1, 1);
        TRANSITION_13.addArcsFromPlace(STATE_2, 1);
        TRANSITION_13.addArcsToPlace(STATE_2, 1);

TRANSITION_13.addArcsFromPlace(com.fakebank.frmvalidation.ATMState.IDL
E, 1);

TRANSITION_13.addArcsToPlace(com.fakebank.frmvalidation.ATMState.POST_
CONDITION_ANY_STATE_OF_1, 1);

        STATE_1.setNoofResources(1);
```

```
    }

    after(Authenticator componentInstance):
        target(componentInstance) &&
        initialization(Authenticator.new())
    {
        componentInstance.initializePetriNet();
    }

    before(Authenticator componentInstance):
        target(componentInstance) &&
        execution(public void auth(Credential))
    {
        if (componentInstance.TRANSITION_12.enabled())
        {
            componentInstance.TRANSITION_12.fire();
        }
        else if (componentInstance.TRANSITION_13.enabled())
        {
            componentInstance.TRANSITION_13.fire();
        }
        else
        {
            throw new RuntimeException(new
com.motorola.compsvalidation.petrinet.PetriNetTransitionNotAllowed());
        }

        indetermHandler.clearAllTransitionsToFire();

indetermHandler.addTransitionToFire(com.fakebank.frmvalidation.ATMStat
e.TRANSITION_7, com.fakebank.frmvalidation.ATMState.IDLE);

indetermHandler.addTransitionToFire(com.fakebank.frmvalidation.ATMStat
e.TRANSITION_8, com.fakebank.frmvalidation.ATMState.AUTHENTICATED);
        indetermHandler.determineAndFireTransition();
    }

    /*
     * Aspect for component KeyboardController
     */
    private Place KeyboardController.STATE_1;
    private Place KeyboardController.STATE_2;

    private Transition KeyboardController.TRANSITION_14;
    private Transition KeyboardController.TRANSITION_15;

    private void KeyboardController.initializePetriNet()
    {
        STATE_1 = new Place("STATE_1");
        STATE_2 = new Place("STATE_2");

        TRANSITION_14 = new Transition("TRANSITION_14");
        TRANSITION_15 = new Transition("TRANSITION_15");

        TRANSITION_14.addArcsFromPlace(STATE_1, 1);
        TRANSITION_14.addArcsToPlace(STATE_2, 1);

TRANSITION_14.addArcsFromPlace(com.fakebank.frmvalidation.ATMState.PRE
```

```
_CONDITION_ANY_STATE, 1);

TRANSITION_14.addArcsToPlace(com.fakebank.frmvalidation.ATMState.IDLE,
1);
        TRANSITION_15.addArcsFromPlace(STATE_2, 1);
        TRANSITION_15.addArcsToPlace(STATE_2, 1);

TRANSITION_15.addArcsFromPlace(com.fakebank.frmvalidation.ATMState.PRE
_CONDITION_ANY_STATE, 1);

TRANSITION_15.addArcsToPlace(com.fakebank.frmvalidation.ATMState.IDLE,
1);

        STATE_1.setNoofResources(1);

    }

    after(KeyboardController componentInstance):
        target(componentInstance) &&
        initialization(KeyboardController.new())
    {
        componentInstance.initializePetriNet();
    }

    before(KeyboardController componentInstance):
        target(componentInstance) &&
        execution(public void cancelOperation())
    {
        if
(com.fakebank.frmvalidation.ATMState.TRANSITION_1.enabled())
        {
            com.fakebank.frmvalidation.ATMState.TRANSITION_1.fire();
        }
        else if
(com.fakebank.frmvalidation.ATMState.TRANSITION_2.enabled())
        {
            com.fakebank.frmvalidation.ATMState.TRANSITION_2.fire();
        }
        else if
(com.fakebank.frmvalidation.ATMState.TRANSITION_3.enabled())
        {
            com.fakebank.frmvalidation.ATMState.TRANSITION_3.fire();
        }
        else
        {
            throw new RuntimeException(new
com.motorola.compsvalidation.petrinet.PetriNetTransitionNotAllowed());
        }

        if (componentInstance.TRANSITION_14.enabled())
        {
            componentInstance.TRANSITION_14.fire();
        }
        else if (componentInstance.TRANSITION_15.enabled())
        {
            componentInstance.TRANSITION_15.fire();
        }
        else
        {
```

```
            throw new RuntimeException(new
com.motorola.compsvalidation.petrinet.PetriNetTransitionNotAllowed());
        }

    }

}
```

ANEXO 3: UM CASO DE TESTE CONTRUÍDO SOB O TAF

```
package com.motorola.testcase.integration.bas.s004;

import com.motorola.taf.framework.BaseTestCase;
import com.motorola.taf.framework.StepNotConfiguredException;
import com.motorola.taf.frontend.option.MultimediaFile;
import com.motorola.taf.frontend.option.MultimediaItem;
import com.motorola.taf.frontend.option.MultimediaScreen;
import com.motorola.taf.frontend.option.PhoneApplication;

public class TC_test extends BaseTestCase
{

    public void buildProcedures() throws StepNotConfiguredException
    {
        // pre: ANY_STATE
        navigationTk.launchApp(PhoneApplication.CAMERA);
        // above statement takes the phone to PHOTO_VIEWFINDER screen
        // post: ANY_STATE

        // pre: PHOTO_VIEWFINDER
        multimediaTk.capturePictureFromCamera();
        // post: SAVE_DISCARD_SCREEN

        // pre: SAVE_DISCARD_SCREEN
        MultimediaFile picture =
multimediaTk.storeMultimediaFileAs(MultimediaItem.STORE_ONLY);
        // post: any state of {PHOTO_VIEWFINDER, VIDEO_VIEWFINDER,
        // BROWSER_APPLICATION}

        // pre: ANY_STATE
        navigationTk.launchApp(PhoneApplication.PICTURES);
        // above statement takes the phone to PICTURES_FILE_LIST
        // screen
        // post: ANY_STATE

        // pre: any state of {SOUNDS_FILE_LIST, PICTURES_FILE_LIST,
        // VIDEOS_FILE_LIST}
        multimediaTk.scrollToAndSelectMultimediaFile(picture);
        // above statement takes the phone to PICTURE_VIEWER screen.
        // post: any state of {SOUND_PLAYER, PICTURE_VIEWER,
        // VIDEO_PLAYER}

        // pre: NONE
        phoneTk.checkScreen(MultimediaScreen.MEDIA_PLAYER);
        // pre: NONE

        // pre: any state of {SOUND_PLAYER, PICTURE_VIEWER,
        // VIDEO_PLAYER}
        multimediaTk.openCurrentFileDetails();
        // post: FILE_DETAILS_SCREEN
```

```
        // pre: FILE_DETAILS_SCREEN
        multimediaTk.verifyAllMultimediaFileDetails(picture);
        // post: FILE_DETAILS_SCREEN

        // SHOULD NOT HAVE THE COMMENTED STATEMENT BELOW; IT WAS
        // PLACED HERE FOR TEST PURPOSES ONLY
        // pre: any state of {SOUNDS_FILE_LIST, PICTURES_FILE_LIST,
        // VIDEOS_FILE_LIST}
        // multimediaTk.deleteFile(picture, true);
        // pre: any state of {SOUNDS_FILE_LIST, PICTURES_FILE_LIST,
VIDEOS_FILE_LIST}

        // pre: ANY_STATE
        phoneTk.returnToPreviousScreen();
        // come back to PICTURE_VIEWER
        // post: ANY_STATE

        // pre: ANY_STATE
        phoneTk.returnToPreviousScreen();
        // come back to PICTURES_FILE_LIST
        // post: ANY_STATE

        // pre: any state of {SOUNDS_FILE_LIST, PICTURES_FILE_LIST,
        // VIDEOS_FILE_LIST}
        multimediaTk.deleteFile(picture, true);
        // pre: any state of {SOUNDS_FILE_LIST, PICTURES_FILE_LIST,
        // VIDEOS_FILE_LIST}

    }

}
```

ANEXO 4: ESPECIFICAÇÃO XML DE ALGUMAS UFs DO TAF

```
<?xml version="1.0" encoding="UTF-8"?>
<systemSpecification
    xmlns="http://www.motorola.com/systemspecification"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.motorola.com/systemspecification
D:\users\rechia\rechia_mestrado\taf_petri11\system_validation\src\java
\com\motorola\systemvalidation\schemas\system_schema.xsd"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xlink:type="extended">

    <systemStates>
        <state xlink:label="IDLE" initial="yes"/>
        <state xlink:label="PHOTO_VIEWFINDER"/>
        <state xlink:label="VIDEO_VIEWFINDER"/>
        <state xlink:label="SAVE_DISCARD_SCREEN"/>
        <state xlink:label="BROWSER_APPLICATION"/>
        <state xlink:label="SOUNDS_FILE_LIST"/>
        <state xlink:label="PICTURES_FILE_LIST"/>
        <state xlink:label="VIDEOS_FILE_LIST"/>
        <state xlink:label="SOUND_PLAYER"/>
        <state xlink:label="PICTURE_VIEWER"/>
        <state xlink:label="VIDEO_PLAYER"/>
        <state xlink:label="FILE_DETAILS_SCREEN"/>
    </systemStates>

    <class
name="com.motorola.taf.utility.function.navigation.api.LaunchApp">
        <methodsRestrictions>
            <method signature="public void execute() throws
com.motorola.taf.framework.ExecutionException">
                <preConditionAnyState/>
                <postConditionAnyState/>
            </method>
        </methodsRestrictions>
        <methodsCallOrder>
            <list type="sequence" quantifier="+">
                <methodCall signature="public void
setApplication(com.motorola.taf.frontend.option.PhoneApplication)"
quantifier="+"/>
                <methodCall signature="public void execute() throws
com.motorola.taf.framework.ExecutionException" quantifier="+"/>
            </list>
        </methodsCallOrder>
    </class>

    <class
name="com.motorola.taf.utility.function.multimedia.api.CapturePictureF
romCamera">
        <methodsRestrictions>
            <method signature="public void execute() throws
com.motorola.taf.framework.ExecutionException">
```

```
                <preConditionState>
                    <state xlink:to="PHOTO_VIEWFINDER"/>
                </preConditionState>
                <postConditionState>
                    <state xlink:to="SAVE_DISCARD_SCREEN"/>
                </postConditionState>
            </method>
        </methodsRestrictions>
        <methodsCallOrder>
            <list type="sequence">
                <methodCall signature="public void execute() throws
com.motorola.taf.framework.ExecutionException" quantifier="+"/>
            </list>
        </methodsCallOrder>
    </class>

    <class
name="com.motorola.taf.utility.function.multimedia.api.StoreMultimedia
FileAs">
        <methodsRestrictions>
            <method signature="public void execute() throws
com.motorola.taf.framework.ExecutionException">
                <preConditionState>
                    <state xlink:to="SAVE_DISCARD_SCREEN"/>
                </preConditionState>
                <postConditionAnyStateOf>
                    <state xlink:to="PHOTO_VIEWFINDER"/>
                    <state xlink:to="VIDEO_VIEWFINDER"/>
                    <state xlink:to="BROWSER_APPLICATION"/>
                </postConditionAnyStateOf>
            </method>
        </methodsRestrictions>
        <methodsCallOrder>
            <list type="sequence">
                <methodCall signature="public void execute() throws
com.motorola.taf.framework.ExecutionException" quantifier="+"/>
            </list>
        </methodsCallOrder>
    </class>

    <class
name="com.motorola.taf.utility.function.multimedia.api.ScrollToAndSele
ctMultimediaFile">
        <methodsRestrictions>
            <method signature="public void execute() throws
com.motorola.taf.framework.ExecutionException">
                <preConditionAnyStateOf>
                    <state xlink:to="SOUNDS_FILE_LIST"/>
                    <state xlink:to="PICTURES_FILE_LIST"/>
                    <state xlink:to="VIDEOS_FILE_LIST"/>
                </preConditionAnyStateOf>
                <postConditionAnyStateOf>
                    <state xlink:to="SOUND_PLAYER"/>
                    <state xlink:to="PICTURE_VIEWER"/>
                    <state xlink:to="VIDEO_PLAYER"/>
                </postConditionAnyStateOf>
            </method>
        </methodsRestrictions>
        <methodsCallOrder>
```

```
            <list type="sequence" quantifier="+">
                <methodCall signature="public void
setItem(com.motorola.taf.frontend.option.MultimediaFile)"
quantifier="+"/>
                <methodCall signature="public void execute() throws
com.motorola.taf.framework.ExecutionException"/>
            </list>
        </methodsCallOrder>
    </class>

    <class
name="com.motorola.taf.utility.function.phone.api.CheckScreen">
        <methodsCallOrder>
            <list type="sequence" quantifier="+">
                <methodCall signature="public void
setTitle(com.motorola.taf.frontend.option.ApplicationScreen)"
quantifier="+"/>
                <methodCall signature="public void execute() throws
com.motorola.taf.framework.ExecutionException" quantifier="+"/>
            </list>
        </methodsCallOrder>
    </class>

    <class
name="com.motorola.taf.utility.function.multimedia.api.OpenCurrentFile
Details">
        <methodsRestrictions>
            <method signature="public void execute() throws
com.motorola.taf.framework.ExecutionException">
                <preConditionAnyStateOf>
                    <state xlink:to="SOUND_PLAYER"/>
                    <state xlink:to="PICTURE_VIEWER"/>
                    <state xlink:to="VIDEO_PLAYER"/>
                </preConditionAnyStateOf>
                <postConditionState>
                    <state xlink:to="FILE_DETAILS_SCREEN"/>
                </postConditionState>
            </method>
        </methodsRestrictions>
        <methodsCallOrder>
            <list type="sequence">
                <methodCall signature="public void execute() throws
com.motorola.taf.framework.ExecutionException" quantifier="+"/>
            </list>
        </methodsCallOrder>
    </class>

    <class
name="com.motorola.taf.utility.function.multimedia.api.VerifyAllMultim
ediaFileDetails">
        <methodsRestrictions>
            <method signature="public void execute() throws
com.motorola.taf.framework.ExecutionException">
                <preConditionState>
                    <state xlink:to="FILE_DETAILS_SCREEN"/>
                </preConditionState>
                <postConditionState>
                    <state xlink:to="FILE_DETAILS_SCREEN"/>
                </postConditionState>
```

```
            </method>
        </methodsRestrictions>
        <methodsCallOrder>
            <list type="sequence" quantifier="+">
                <methodCall signature="public void
setFile(com.motorola.taf.frontend.option.MultimediaFile)"/>
                <methodCall signature="public void execute() throws
com.motorola.taf.framework.ExecutionException" quantifier="+"/>
            </list>
        </methodsCallOrder>
    </class>

    <class
name="com.motorola.taf.utility.function.phone.api.ReturnToPreviousScre
en">
        <methodsRestrictions>
            <method signature="public void execute() throws
com.motorola.taf.framework.ExecutionException">
                <preConditionAnyState/>
                <postConditionAnyState/>
            </method>
        </methodsRestrictions>
        <methodsCallOrder>
            <list type="sequence">
                <methodCall signature="public void execute() throws
com.motorola.taf.framework.ExecutionException" quantifier="+"/>
            </list>
        </methodsCallOrder>
    </class>

    <class
name="com.motorola.taf.utility.function.multimedia.api.DeleteFile">
        <methodsRestrictions>
            <method signature="public void execute() throws
com.motorola.taf.framework.ExecutionException">
                <preConditionAnyStateOf>
                    <state xlink:to="SOUNDS_FILE_LIST"/>
                    <state xlink:to="PICTURES_FILE_LIST"/>
                    <state xlink:to="VIDEOS_FILE_LIST"/>
                </preConditionAnyStateOf>
                <postConditionAnyStateOf>
                    <state xlink:to="SOUNDS_FILE_LIST"/>
                    <state xlink:to="PICTURES_FILE_LIST"/>
                    <state xlink:to="VIDEOS_FILE_LIST"/>
                </postConditionAnyStateOf>
            </method>
        </methodsRestrictions>
        <methodsCallOrder>
            <list type="sequence" quantifier="+">
                <methodCall signature="public void
setFile(com.motorola.taf.frontend.option.MultimediaFile)"
quantifier="+"/>
                <methodCall signature="public void execute() throws
com.motorola.taf.framework.ExecutionException"/>
            </list>
        </methodsCallOrder>
    </class>

</systemSpecification>
```

REFERÊNCIAS BIBLIOGRÁFICAS

AHO, A.; SETHI, R.; ULLMAN, J. **Compilers: Principles, Techniques and Tools**. Redwood City: Addison-Wesley, 1986. 2ª edition.

BOSCH, J.; MOLIN, P.; MATTSON, M. Framework Problems and Experiences. In: **Building Application Frameworks.** New York:Wiley, 1999. p. 55-82.

CAMPO, M. **Compreensão virtual de frameworks através da introspecção de exemplos**. Porto Alegre: Instituto de Informática da UFRGS, 1997. Tese de Doutorado.

CARDOSO, J; VALETTE, R. **Redes de Petri**. Florianópolis, Brasil: Editora da UFSC, 1997.

CLARK, J. **XSL Transformations, XSLT, Specification 1.0**. W3C Recommendation. 2003. Disponível na Internet em http://www.w3.org/TR/xslt. Acessado em 12 de setembro de 2005.

CUNHA, R. **Suporte à análise de compatibilidade comportamental e estrutural entre componentes no ambiente SEA.** Florinópolis: Departamento de Informática e Estatística da UFSC, 2005. Dissertação de mestrado.

ECLIPSE, Eclipse Foundation. **Eclipse.org Main Page.** 2005. Disponível na Internet em http://eclipse.org/. Acessado em 19 de novembro de 2005.

ECLIPSE, Eclipse Foundation. **The AspectJ project at Eclipse.org**. 2005. Disponível na Internet em http://eclipse.org/aspectj/. Acessado em 3 de outubro de 2005.

ELRAD, T.; FILMAN, R.; BADER, A. Aspect-oriented programming: Introduction. **Communications of the ACM**. Volume 44, Número 10, p. 29-32, outubro de 2001.

FAYAD, M.; SCHMIDT, D.; JOHNSON, R. Application frameworks. In: **Building Application Frameworks.** New York:Wiley, 1999. p. 1-27.

FONTOURA, M.; BRAGA, C.; MOURA, L.; LUCENA, J. Using domain specific languages to instantiate object-oriented frameworks. **IEE Proceedings-Software**. Volume 147, Número 4, p. 109-116, agosto de 2000.

GAMMA, E.; HELM, R.; JOHNSON, R. et al. **Design patterns: elements of reusable object-oriented software.** Reading: Addison-Wesly, 1994.

HOU, D.; HOOVER, J. Towards Specifying Constraints for Object-Oriented Frameworks. In: *Conference of the Centre For Advanced Studies on Collaborative Research,* Toronto, Ontario, Canada, 05-07 de novembro de 2001. **Proceedings…** Ontario: IBM Press, p. 5.

JOHNSON, R.; FOOTE, B. Designing reusable classes. **Journal of Object-Oriented Programming**. Volume 1, Número 2, p. 22-35, junho/julho de 1988.

JÜNGEL, M.; KINDLER, E.; WEBER, M. The Petri Net Markup Language. **Petri Net Newsletter**. Volume 59, p. 24-29, 2000.

KICZALES, G.; HILSDALE, E.; HUGUNIN, J. *et al*. Getting started with ASPECTJ. **Communications of ACM**. Volume 44, Número 10, p. 59-65, outubro de 2001.

LAM, P.; KUNCAK, V.; RINARD, M. Crosscutting Techniques in Program Specification and analysis. In: *4th international Conference on Aspect-Oriented Software Development* (AOSD '05), Chicago, Illinois, 14–18 de março 2005. **Proceedings…** New York, NY: ACM Press, p. 169-180.

MILI, H.; MILI, F.; MILI, A. Reusing Software: Issues and Research Directions. **IEEE Transactions on Software Engineering**. Volume 21, Número 6, p. 528-562, 1995.

MOSER, S.; NIERSTRASZ, O. The effect of object-oriented frameworks on developer productivity. **IEEE Computer**. Volume 29, Número 9, p. 45-51, 1993.

MURRAY, L.; CARRINGTON, D.; STROOPER, P. An Aproach to specifying software frameworks. In: *27th Conference on Australasian Computer Science,* Dunedin, Nova Zelândia. **Proceedings…** Darlinghurst, Australia: Australian Computer Society, Inc, p. 185-192.

NASA, The National Aeronautics and Space Administration. **Java PathFinder.** 1999. Disponível na Internet em http://javapathfinder.sourceforge.net/. Lido em 14 de novembro de 2005.

OLIVEIRA, T.; ALENCAR, P.; FILHO, I. e*t al*. Software Process Representation and Analysis for Framework Instantiation. **IEEE Transactions on Software Engineering**. Volume 30, Número 3, p. 145-159, Março de 2004.

OMG, Object Management Group. **Unified Modeling Language**, 1997. Disponível na internet em http://www.uml.org/. Acessado em 09 de setembro de 2005.

SILVA, R. **Suporte ao Desenvolvimento e uso de *Frameworks* e Componentes.** Porto Alegre: Instituto de Informática da UFRGS, 2000. Tese de Doutorado.

SILVA, R.; FREIBERGER, E. Helping Object-Oriented Framework Use and Evaluation by means of Historical Use information. In: *19th International Conference on Automated Software Engineering* (ASE 2004), Linz, Austria, 20-25 de setembro de 2004. **Proceedings…** Linz: IEEE Computer Society, p. 278-281.

SILVA, R.; PRICE, R. Component interface pattern. In: *9th Conference on Pattern Language of Programs (PLOP 2002)*. **Proceedings…** Monticello: sep. 2002. 17p.

STACY, W.; HELM, R.; KAISER, G.. Panel: Ensuring semantic integrity of reusable objects. In: *Conference on Pattern Language of Programs (OOPSLA'92)*, Vancouver, Canadá, 1992. **Proceedings…** p. 298-302.

SUN, Sun Developer Network. **Java Technology and Web Services**, 1994. Disponível na Internet em http://java.sun.com/webservices/. Acessado em 07 de novembro de 2005.

THANH, C; KLAUDEL, H. Object-oriented modelling with high-level modular Petri nets. In: *4th International Conference (IFM 2004), Canterbury, UK, 4-7 de abril de 2004*. **Proceedings…** Canterbury: Springer-Verlag. Volume 2988 of Lecture Notes in Computer Science, p. 287-306.

UBAYASHI, N.; TETSUO, T. **Aspect-Oriented Programming with Model Checking.** In: *1st international Conference on Aspect-Oriented Software Development (*AOSD '02*),* Enschede, The Netherlands, 22-26 de abril de 2002. **Proceedings…** New York: ACM Press, p. 148-154.

W3C, World Wide Web Consortium. **Extensible Markup Language**. 1996. Disponível na Internet em http://www.w3.org/XML/. Acessado em 12 de setembro de 2005.

W3C, World Wide Web Consortium. **Namespaces in XML**. 1999. Disponível na Internet em http://www.w3.org/TR/1999/REC-xml-names-19990114/. Acessado em 12 de setembro de 2005.

W3C, World Wide Web Consortium. **XML Schema**. 2000. Disponível na Internet em http://www.w3.org/XML/Schema. Acessado em 12 de setembro de 2005.

XEROX, Xerox Corporation. **The AspectJ Language**. 1998. Disponível na Internet em http://www.eclipse.org/aspectj/doc/released/progguide/language.html. Acessado em 03 de outubro de 2005.